QUIC Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 12, 2017

M. Bishop
Microsoft

February 8, 2017

Header Compression for HTTP/QUIC
draft-bishop-quic-http-and-qpack-02

Abstract

HTTP/2 [RFC7540] uses HPACK [RFC7541] for header compression. However, HPACK relies on the in-order message-based semantics of the HTTP/2 framing layer in order to function. Messages can only be successfully decoded if processed by the decoder in the same order as generated by the encoder. This draft refines HPACK to loosen the ordering requirements for use over QUIC [I-D.ietf-quic-transport].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 12, 2017.

Copyright Notice

Table of Contents

1. Introduction

HPACK has a number of features that were intended to provide performance advantages to HTTP/2, but which don't live well in an out-of-order environment such as that provided by QUIC.

The largest challenge is the fact that elements are referenced by a very fluid index. Not only is the index implicit when an item is added to the header table, the index will change without notice as other items are added to the header table. Static entries occupy the first 61 values, followed by dynamic entries. A newly-added dynamic entry would cause older dynamic entries to be evicted, and the retained items are then renumbered beginning with 62. This means that, without processing all preceding header sets, no index into the dynamic table can be interpreted, and the index of a given entry cannot be predicted.

Any solution to the above will almost certainly fall afoul of the memory constraints the decompressor imposes. The automatic eviction of entries is done based on the compressor's declared dynamic table

size, which MUST be less than the maximum permitted by the decompressor (and relayed using an HTTP/2 SETTINGS value).

Further, streams in QUIC are lossy in the presence of stream resets. While HTTP/2 (via TCP) guarantees the delivery of all previously-sent data on a stream even if that stream is reset, QUIC does not retransmit lost frames if a stream has been reset, and may discard data which has not yet been delivered to the application.

Previous versions of QPACK were small deltas of HPACK to introduce order-resiliency. This version departs from HPACK more substantially to add resilience against reset message streams.

In the following sections, this document proposes a new version of HPACK which makes different trade-offs, enabling partial out-of-order interpretation and bounded memory consumption with minimal head-of-line blocking. None of the proposed improvements to HPACK (strongly-typed fields, binary compression of common header syntax) are currently included, but certainly could be.

1.1. Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, [RFC2119] and indicate requirement levels for compliant implementations.

2. QPACK

2.1. Basic model

HPACK combines header table modification and message header emission in a single sequence of coded bytes. QPACK bifurcates these into two channels:

- A connection-wide series of table update instructions sent on a dedicated headers stream
- Non-modifying instructions which use the current header table state to encode message headers

Because the per-message instructions introduce no changes to the header table state, no state is lost if these instructions are discarded due to a stream reset.

2.2. Changes to Static and Dynamic Tables

QPACK uses two tables for associating header fields to indexes. The static table is unchanged from [RFC7541].

The dynamic table is a map from index to header field. Indices are arbitrary numbers greater than the last index of the static table and less than 2^27. Each insert instruction will specify the index being modified. While any index MAY be chosen for a new entry, smaller numbers will yield better compression performance.

The dynamic table is still constrained to the size specified by the decoder. An attempt to add a header to the dynamic table which causes it to exceed the maximum size MUST be treated as an error by a decoder. To enable encoders to reclaim space, encoders can delete entries in the dynamic table, but can only reuse the index or the space after receiving confirmation of a successful deletion.

Because it is possible for QPACK frames to arrive which reference indices which have not yet been defined, such frames MUST wait until another frame has arrived and defined the index. In order to guard against malicious peers, implementations SHOULD impose a time limit and treat expiration of the timer as a decoding error. However, if the implementation chooses not to abort the connection, the remainder of the header block MUST be decoded and the output discarded.

2.2.1. Changes to Header Table Size

HTTP/QUIC prohibits mid-stream changes of settings. As a result, only one table size change is possible: From the value a client assumes during the 0-RTT flight to the actual value included in the server's SETTINGS frame. The assumed value is required to be either a server's previous value or zero. A server whose configuration has recently changed MAY overlook inadvertent violations of its maximum table size during the first round-trip.

In the case that the value has increased, either from zero to a non-zero value or from the cached value to a higher value, no action is required by the client. The encoder can simply begin using the additional space. In the case that the value has decreased, the encoder MUST immediately emit delete instructions which, upon completion, would bring the table within the required size.

Regardless of changes to header table size, the encoder MUST NOT add entries to the table which would result in a size greater than the maximum permitted. This can imply that no additions are permitted while waiting for these delete instructions to complete.

2.2.2. Dynamic Table State Synchronization

In order to ensure table consistency, all modifications of the header table occur on a dedicated control stream. Message control streams contain only indexed and literal header entries.

No entries are automatically evicted from the dynamic table. Size management is purely the responsibility of the encoder, which MUST NOT exceed the declared memory size of the decoder.

The encoder SHOULD track the following information about each entry in the table:

- The list of recently-active streams which reference the entry in a trailer block, if any
- The list of recently-active streams which reference the entry in a non-trailer block, if any

"Recently-active" streams are those which are still open or were closed less than a reasonable number of RTTs ago. An implementation MAY vary its definition of "recent" to trade off memory consumption and timely completion of deletes.

The encoder MUST consider memory as committed beginning when the indexed entry is assigned.

When the encoder wishes to delete an inserted value, it flows through the following set of states:

1. *Delete requested.* The encoder emits a delete instruction indicating which streams might have referenced the entry. The encoder MUST NOT reference the entry in any subsequent frame until this state machine has completed and MUST continue to include the entry in its calculation of consumed memory.
2. *Delete pending.* The decoder receives the delete instruction and checks the current state of its incoming streams (see Section 2.3.2.2). If more references might arrive, it stores the streams still needed and waits for them to complete.
3. *Delete acknowledged.* The decoder has received all QPACK frames which reference the deleted value, and can safely delete the entry. The decoder SHOULD promptly emit a Delete-Ack instruction on the header management stream.
4. *Delete completed.* When the encoder receives a Delete-Ack instruction acknowledging the delete, it no longer counts the

size of the deleted entry against the table size and MAY emit insert instructions for the field with a new value.

2.3. Format of Header Management stream

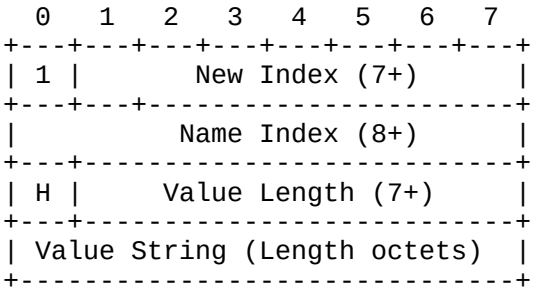
The header management stream contains a series of QPACK instructions with no message boundaries. Data on this stream SHOULD be processed as soon as it arrives.

This section describes the instructions which are possible on the Header Management stream.

2.3.1. Insert

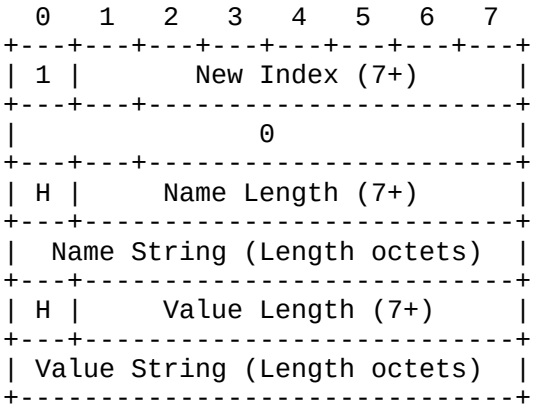
An addition to the header table starts with the '1' one-bit pattern, followed by the new index of the header represented as an integer with a 7-bit prefix. This value is always greater than the number of entries in the static table.

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the index of the entry is represented as an integer with an 8-bit prefix (see Section 5.1 of [RFC7541]). This value is always non-zero.



Insert Header Field -- Indexed Name

Otherwise, the header field name is represented as a string literal (see Section 5.2 of [RFC7541]). A value 0 is used in place of the 8-bit index, followed by the header field name.



Insert Header Field -- New Name

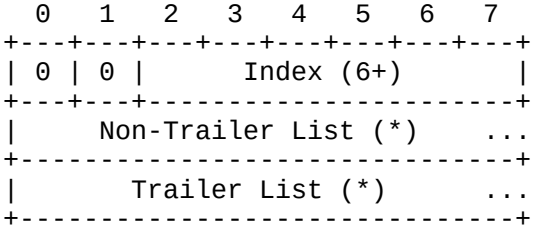
Either form of header field name representation is followed by the header field value represented as a string literal (see Section 5.2 of [RFC7541]).

An encoder MUST NOT attempt to place a value at an index not known to be vacant. A decoder MUST treat the attempt to insert into an occupied slot as a fatal error.

2.3.2. Delete

A deletion from the header table starts with the ’00’ two bit pattern, followed by the index of the affected entry represented as an integer with a 6-bit prefix. This value is always greater than the number of entries in the static table.

A delete instruction then encodes a series of stream IDs which might have contained references to the entry in question.



Delete Instruction

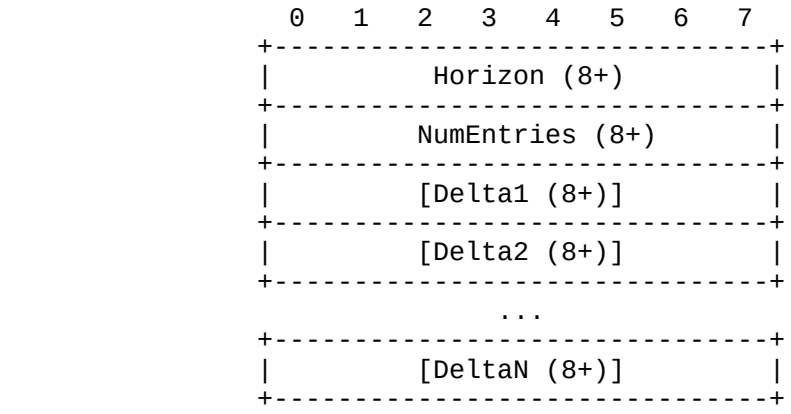
Both the Non-Trailer List and Trailer List are Stream ID Lists (see below) encoding a list of streams which might have referenced the entry either in non-trailer or trailer blocks.

2.3.2.1. Stream ID List

A Stream ID List encodes a sequence of stream IDs in two parts: First, a Horizon value indicates the first non-occurrence about which data is maintained. If data is maintained from the beginning of the connection, the Horizon is zero. This allows senders to succinctly express both old state which has been discarded and large regions where many or all streams contain references.

Following the horizon, a sequence of deltas indicates all streams since the Horizon on which a value has been used.

In the simplest case, a Stream ID List might be a horizon value followed by one zero byte. This indicates an absolute cut-off after which the entry is guaranteed not to be referenced.



Stream ID List

The field are as follows:

Horizon: The ID of the first stream for which the sender retains state which does not reference the deleted entry in the indicated block

NumEntries: The number of streams greater than the Horizon which might reference the entry and are listed in the remainder of the instruction

Delta1..N: A sequence of streams greater than the Horizon which might reference the entry, encoded as the difference in stream number from the previously-listed stream. This field is repeated NumEntries times.

2.3.2.2. Delete Validation

In order to safely delete an entry, a decoder MUST ensure that all outstanding references have arrived and been processed. Because no data is available about stream IDs less than the Horizon, a decoder MUST assume that any earlier stream ID might have contained a reference to the value in question.

A decoder can ensure all outstanding references have been processed by verifying that the following statements are true:

- In the Non-Trailer Block, all streams less than the Horizon and all streams explicitly listed are in one of two states:
  - closed
  - headers completely processed
- In the Trailer Block, all streams less than the Horizon and all streams explicitly listed are in one of three states:
  - closed
  - headers completely processed AND no trailers are expected
  - trailers completely processed

An implementation MAY omit the "trailers completely processed" case, since the stream is expected to close immediately after receipt of the trailers block.

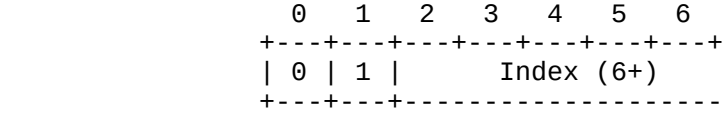
If these conditions are not met upon receipt of a Delete instruction, a decoder MUST wait to emit a Delete-Ack instruction until the outstanding streams have reached an appropriate state.

Note that a decoder MAY condense the list of specified streams by increasing the Horizon value and discarding those explicitly-listed stream IDs which are less than the new Horizon it has chosen. This delays delete completion, but reduces the amount of state to be tracked by the decoder without changing the correctness of the requirements above.

2.3.3. Delete-Ack

Confirmation that a delete has completed is expressed by an instruction which starts with the '01' two-bit pattern, followed by the index of the affected entry represented as an integer with a 6-bit prefix. This value is always greater than the number of entries in the static table.

Note that unlike all other instructions, this instruction refers to the receiver's dynamic table, not the sender's.



Delete-Ack Instruction

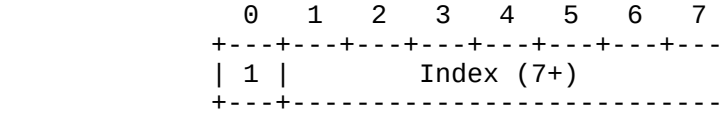
This instruction MUST NOT be sent before the conditions described in Section 2.3.2.2 have been satisfied, and SHOULD be sent as soon as possible once they are.

2.4. Format of Encoded Headers on Message Streams

Frames which carry HTTP message headers encode them using the following instructions:

2.4.1. Indexed Header Field Representation

An indexed header field representation identifies an entry in either the static table or the dynamic table and causes that header field to be added to the decoded header list, as described in Section 3.2 of [RFC7541].



Indexed Header Field

An indexed header field starts with the '1' 1-bit pattern, followed by the index of the matching header field, represented as an integer with a 7-bit prefix (see Section 5.1 of [RFC7541]).

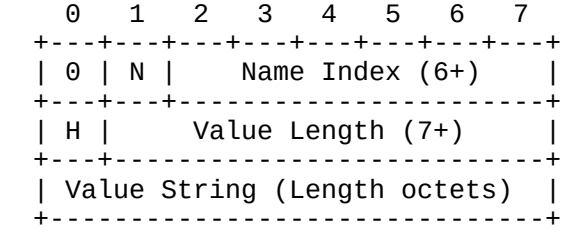
The index value of 0 is not used. It MUST be treated as a decoding error if found in an indexed header field representation.

2.4.2. Literal Header Field Representation

A literal header field representation starts with the '0' 1-bit pattern and causes a header field to be added the decoded header list.

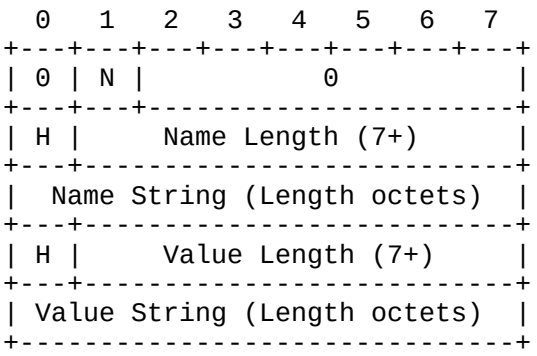
The second bit, 'N', indicates whether an intermediary is permitted to add this header to the dynamic header table on subsequent hops. When the 'N' bit is set, the encoded header MUST always be encoded with this specific literal representation. In particular, when a peer sends a header field that it received represented as a literal header field with the 'N' bit set, it MUST use the same representation to forward this header field. This bit is intended for protecting header field values that are not to be put at risk by compressing them (see Section 7.1 of [RFC7541] for more details).

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the index of the entry is represented as an integer with a 6-bit prefix (see Section 5.1 of [RFC7541]). This value is always non-zero.



Literal Header Field -- Indexed Name

Otherwise, the header field name is represented as a string literal (see Section 5.2 of [RFC7541]). A value 0 is used in place of the 6-bit index, followed by the header field name.



Literal Header Field -- Literal Name

Either form of header field name representation is followed by the header field value represented as a string literal (see Section 5.2).

3. Use in HTTP/QUIC

HTTP/QUIC [I-D.ietf-quic-http] currently retains the HPACK encoder/decoder from HTTP/2, using a Sequence number to enforce ordering. Using QPACK instead would entail the following changes:

- The Sequence field is removed from HEADERS frames (Section 5.2.2) and PUSH_PROMISE frames (Section 5.2.6).
- Header Block Fragments consist of QPACK data instead of HPACK data.
- An additional control stream is reserved for header table updates. Alternately, this could be carried by HEADERS frames on the connection control stream.

A HEADERS or PUSH_PROMISE frame MAY contain an arbitrary number of QPACK instructions, but QPACK instructions SHOULD NOT cross a boundary between successive HEADERS frames. A partial HEADERS or PUSH_PROMISE frame MAY be processed upon arrival and the resulting partial header set emitted or buffered according to implementation requirements.

4. Performance Considerations

While QPACK is designed to minimize head-of-line blocking between streams on header decoding, there are some situations in which lost or delayed packets can still impact the performance of header compression.

References to indexed entries will block if the frame containing the entry definition is lost or delayed. Encoders MAY choose to trade off compression efficiency and avoid blocking by using literal instructions rather than referencing the dynamic table until the insertion is believed to be complete.

Since it is possible to insert header values without emitting them on a stream, an encoder MAY proactively insert header values which it believes will be needed on future requests.

Delayed frames which prevent deletes from completing can prevent the encoder from adding any new entries due to the maximum table size. This does not block the encoder from continuing to make requests, but could sharply limit compression performance. Encoders would be well-served to delete entries in advance of encountering the table maximum. Decoders SHOULD be prompt about emitting Delete-Ack instructions to enable the encoder to recover the table space.

5. Security Considerations

A malicious encoder might attempt to consume a large amount of space on the decoder by opening the maximum number of streams, adding entries to the table, then sending delete instructions enumerating many streams in a Stream ID List.

To guard against such attacks, a decoder SHOULD bound its state tracking by generalizing the list of streams to be tracked. This is most easily achieved by advancing the Horizon to a later value and discarding explicit Stream IDs to track, but can also be accomplished by eliding explicit streams in ranges. This does not cause any loss of consistency for deletes, but could delay completion and reduce performance if done aggressively.

6. IANA Considerations

This document currently makes no request of IANA, and might not need to.

7. Acknowledgements

This draft draws heavily on the text of [RFC7541]. The indirect input of those authors is gratefully acknowledged, as well as ideas gleefully stolen from:

- Jana Iyengar
- Patrick McManus

- Martin Thomson
- Charles 'Buck' Krasic
- Kyle Rose

8. Normative References

[I-D.ietf-quic-http]
Bishop, M., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-01 (work in progress), January 2017.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <http://www.rfc-editor.org/info/rfc7540>.

[RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <http://www.rfc-editor.org/info/rfc7541>.

Author's Address

Mike Bishop
Microsoft

Email: michael.bishop@microsoft.com

QUIC Working Group
Internet-Draft
Intended status: Standards Track
Expires: June 17, 2018

M. Bishop
Akamai

December 14, 2017

Header Compression for HTTP/QUIC
draft-bishop-quic-http-and-qpack-07

Abstract

HTTP/2 [RFC7540] uses HPACK [RFC7541] for header compression. However, HPACK relies on the in-order message-based semantics of the HTTP/2 framing layer in order to function. Messages can only be successfully decoded if processed by the decoder in the same order as generated by the encoder. This draft refines HPACK to loosen the ordering requirements for use over QUIC [I-D.ietf-quic-transport].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on June 17, 2018.

Table of Contents

1. Introduction

HPACK has a number of features that were intended to provide performance advantages to HTTP/2, but which don’t live well in an out-of-order environment such as that provided by QUIC.

The largest challenge is the fact that elements are referenced by a very fluid index. Not only is the index implicit when an item is added to the header table, the index will change without notice as other items are added to the header table. Static entries occupy the first 61 values, followed by dynamic entries. A newly-added dynamic entry would cause older dynamic entries to be evicted, and the retained items are then renumbered beginning with 62. This means that, without processing all preceding header sets, no index into the dynamic table can be interpreted, and the index of a given entry cannot be predicted.

Any solution to the above will almost certainly fall afoul of the memory constraints the decompressor imposes. The automatic eviction of entries is done based on the compressor's declared dynamic table size, which MUST be less than the maximum permitted by the decompressor (and relayed using an HTTP/2 SETTINGS value).

Further, streams in QUIC are lossy in the presence of stream resets. While HTTP/2 (via TCP) guarantees the delivery of all previously-sent data on a stream even if that stream is reset, QUIC does not retransmit lost frames if a stream has been reset, and may discard data which has not yet been delivered to the application.

Early versions of QPACK were small deltas of HPACK to introduce order-resiliency. Recent versions depart from HPACK more substantially to add resilience against reset message streams and reduce the impact of head-of-line blocking.

In the following sections, this document proposes a successor to HPACK which makes different trade-offs, enabling partial out-of-order interpretation and bounded memory consumption with minimal head-of-line blocking. None of the proposed improvements to HPACK (strongly-typed fields, binary compression of common header syntax) are currently included, but certainly could be.

1.1. Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, [RFC2119] and indicate requirement levels for compliant implementations.

2. QPACK - Concepts

HPACK combines header table modification and message header emission in a single sequence of coded bytes. QPACK bifurcates these into three channels:

- Connection-wide sets of table update instructions sent on non-request streams
- Connection-wide feedback on stream and checkpoint state on a single non-request stream
- Non-modifying instructions which use the current header table state to encode message headers on request streams

Because the per-message instructions introduce no changes to the header table state, no state is lost if these instructions are discarded due to a stream reset. Because the updates to the header table supply their own order controls (the checkpoint logic), they can be processed in any order and therefore delivered as messages using unidirectional QUIC streams.

2.1. Changes to Static and Dynamic Tables

QPACK uses two tables for associating header fields to indexes. The static table is unchanged from [RFC7541]. Unlike in [RFC7541], the tables are not concatenated, but are referenced separately.

The dynamic table is a map from index to header field. Indices are arbitrary numbers between 1 and 2^27. Each insert instruction will specify the index being modified. While any index MAY be chosen for a new entry, smaller numbers will yield better compression performance.

With decoder consent (see Section 4.1), it is possible for QPACK instructions to arrive which reference indices which have not yet been defined. Such instructions MUST wait until the index definition has arrived. In order to guard against malicious peers, implementations supporting blocking SHOULD impose a time limit and treat expiration of the timer as a decoding error.

2.1.1. Dynamic Table State Synchronization

In order to ensure table consistency, all modifications of the header table occur as separate messages rather than on request streams. Request streams contain only indexed and literal header entries.

No entries are automatically evicted from the dynamic table. Size management is purely the responsibility of the encoder, which MUST NOT exceed the declared memory size of the decoder.

To simplify state management in the dynamic table, _checkpoints_ are introduced. A checkpoint is used to track entries added to the dynamic table and streams that reference those entries, rather than

maintaining the full state of which streams reference which table entries.

Checkpoints are unordered and have an identifier which MUST be unique among checkpoints which have not been dropped. Each checkpoint has a unidirectional stream which begins with its identifier and contains a series of updates associated with that checkpoint. These updates SHOULD be processed as they arrive; it is not necessary (and might not be desirable) to wait for all instructions associated with a checkpoint to arrive before beginning to process it.

The feedback stream is used to relay state transitions to the peer. For example, when a decoder is done processing a header block, it signals this using the HEADERS_DONE message. The encoder uses this information to track which checkpoints can be dropped.

2.1.1.1. Checkpoint Lifecycle

A checkpoint is created by opening a new checkpoint stream. This places the checkpoint in the NEW state for both encoder and decoder. The encoder typically has at least one checkpoint in the NEW state.

Flushing a checkpoint is a two-step operation. First, the checkpoint stream is closed. At that time, the encoder’s NEW checkpoint becomes PENDING. The decoder moves its NEW checkpoint directly to LIVE and responds with an ACK_FLUSH message on the feedback stream. When the encoder receives this message, its PENDING checkpoint becomes LIVE.

Unused entries are evicted indirectly, by dropping checkpoints. Before a checkpoint can be dropped, its state is changed to DYING. Changing a checkpoint’s state to DYING allows the checkpoint to age out. This is a strictly internal state on the encoder, and not visible to the decoder. A DYING checkpoint can be returned to LIVE at the encoder’s discretion if necessary.

The encoder can change a DYING checkpoint to DEAD (sending a DROP instruction) when it is no longer referenced by any outstanding header blocks. The encoder sends the DROP command to the decoder when it declares a checkpoint DEAD.

To ensure consistency, the decoder drops the corresponding checkpoint and responds with an ACK_DROP message only when it has fully received all instructions the encoder has issued up to that point. The encoder drops the DEAD checkpoint upon receipt of the ACK_DROP message.

When a checkpoint is dropped by encoder or decoder, the table entries it references are checked: if an entry is no longer referenced by any checkpoint, the entry is evicted.

Dropping a checkpoint and the entries associated with it is not limited to just the oldest checkpoint; any DYING checkpoint - as long as state transition rules are followed - may be dropped. This flexibility permits the encoder to use a number of strategies for entry eviction.

As long as the maximum dynamic table size is observed, new checkpoints can be created; no upper limit on the number of checkpoints is specified. A well-balanced spread of checkpoints permits the encoder to recycle entries effectively.

2.2. Encoding Constraints

2.2.1. Permitted References

When encoding headers on a request stream, an encoder MAY reference any static table entry or any dynamic header table entry referenced by a LIVE checkpoint. References to entries in NEW or PENDING checkpoints are permitted only if the client has set "SETTING_QPACK_BLOCKING_PERMITTED" (see Section 4.1).

If a decoder receives a reference to an empty slot in the dynamic table but has not sent "SETTING_QPACK_BLOCKING_PERMITTED", this MUST be treated as a stream error of type "ERROR_QPACK_INVALID_REFERENCE" if on a request stream. References to empty slots in the dynamic table on a checkpoint stream MUST be treated as a connection error of type "ERROR_QPACK_INVALID_REFERENCE".

References to DYING checkpoints are possible by returning the checkpoint to LIVE, but this is usually inadvisable. Table entries contained only in a DEAD checkpoint can never be referenced.

2.2.2. Header Table Size

As in HPACK, the dynamic table is constrained to the maximum size specified by the decoder. An attempt to add a header to the dynamic table or to create a new checkpoint which causes it to exceed the maximum size MUST be treated as an error by a decoder. To enable encoders to reclaim space, encoders can drop old checkpoints (see Section 2.1.1).

The total table size is calculated as follows:

- The size of each entry is calculated as in HPACK

- Each checkpoint that has not been removed, regardless of state, consumes 64 bytes

2.2.2.1. Table Size Changes

HTTP/QUIC prohibits mid-stream changes of settings. As a result, only one table size change is possible: From the value a client assumes during the 0-RTT flight to the actual value included in the server's SETTINGS frame. The assumed value is required to be either a server's previous value or zero. A server whose configuration has recently changed MAY overlook inadvertent violations of its maximum table size during the first round-trip.

In the case that the value has increased, either from zero to a non-zero value or from the cached value to a higher value, no action is required by the client. The encoder can simply begin using the additional space. In the case that the value has decreased, the encoder MUST move checkpoints to the DYING state which, upon removal, would bring the table within the required size.

Regardless of changes to header table size, the encoder MUST NOT create new checkpoints or add entries to the table which would result in a size greater than the maximum permitted. This can imply that no additions are permitted while waiting for old checkpoints to complete.

3. Wire Format

QPACK instructions occur on three stream types, each of which uses a separate instruction space.

The feedback stream is a bidirectional server-initiated stream used for acknowledgement of actions and checkpoint state management. Checkpoint streams are unidirectional streams from encoder to decoder. Both types of streams consist of a series of QPACK instructions with no message boundaries, preceded by a stream header for checkpoint streams.

Finally, the contents of HEADERS and PUSH_PROMISE frames on request streams reference the QPACK table state.

This section describes the instructions which are possible on each stream type.

3.1. Feedback Stream

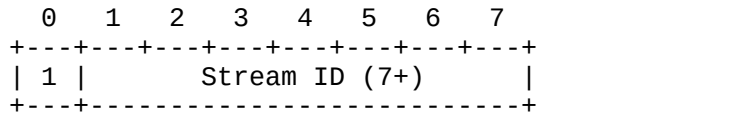
Stream 1, the first server-initiated bidirectional stream, is used as the feedback stream, since the client does not need to begin sending data on this stream until it has received data from the server.

This stream is critical to the HTTP/QUIC connection, and carries a stream of the instructions defined in this section. Data on this stream SHOULD be processed as soon as it arrives.

3.1.1. HEADERS_DONE

When the decoder has processed a frame containing header emission instructions (Section 3.3, HEADERS or PUSH_PROMISE frames) on a stream, it MUST emit a HEADERS_DONE message on the feedback stream. The same Stream ID can be identified multiple times, as multiple header-containing blocks can be sent on a single stream in the case of intermediate responses, trailers, pushed requests, etc.

Since header frames on a request stream are received and processed in order, this gives the encoder precise feedback on which header blocks within a stream have been fully processed. This information can then be used to correctly track outstanding stream references to checkpoints.

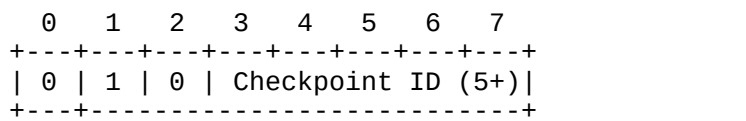


HEADERS_DONE instruction

3.1.2. ACK_FLUSH

When the decoder has finished processing all instructions that make up a checkpoint, it MUST indicate successful processing to the encoder by emitting an ACK_FLUSH instruction on the feedback stream.

Upon emitting an ACK_FLUSH, the checkpoint transitions from NEW to LIVE on the decoder. Upon receipt of an ACK_FLUSH, the checkpoint transitions from PENDING to LIVE on the encoder.

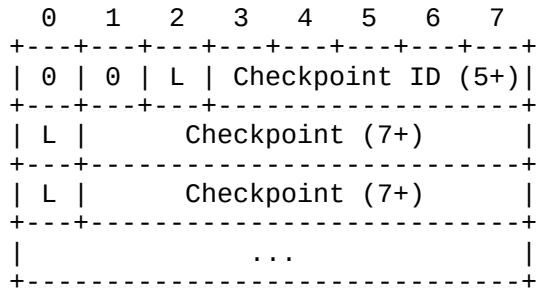


ACK_FLUSH instruction

3.1.3. DROP

When an encoder has received sufficient HEADERS_DONE messages to know that a DYING checkpoint has no outstanding references, it emits a DROP instruction to inform the decoder that the checkpoint can be removed. Upon sending a DROP instruction, a DYING checkpoint becomes DEAD. The DROP instruction also includes the IDs of any PENDING or NEW checkpoints which reference entries contained in the checkpoint being dropped. The "L" bit in each byte indicates whether another checkpoint ID follows (L=0) or this is the final byte of the DROP instruction (L=1).

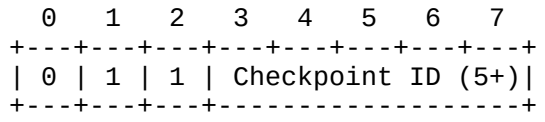
Upon receiving a DROP instruction, if all listed checkpoints have been fully processed (transitioned from NEW to LIVE), the identified LIVE checkpoint is immediately removed from the decoder state and an ACK_DROP instruction is emitted. Otherwise, the decoder saves the DROP instruction until other checkpoints become LIVE.



DROP instruction

3.1.4. ACK_DROP

When a decoder receives a DROP instruction, it removes the referenced checkpoint from its state and clears any table entries which were referenced only by that checkpoint. It then emits an ACK_DROP instruction. When an encoder receives an ACK_DROP instruction, it removes the corresponding DEAD checkpoint from its state and clears any table entries which were referenced only by that checkpoint.



ACK_DROP instruction

3.2. Checkpoint Streams

Each checkpoint stream indicates the creation and content of a NEW checkpoint. Each checkpoint has an ID; these IDs are chosen arbitrarily by the encoder, though lower values SHOULD be preferred. IDs of checkpoints which have been dropped MAY be reused for future NEW checkpoints.

When the encoder has finished writing all data on the stream, it changes the checkpoint to PENDING. When the decoder has received and processed all data on the stream, it changes the checkpoint to LIVE and generates an ACK_FLUSH.

Unidirectional streams in HTTP/QUIC begin with a stream header indicating the nature of the stream content; the identifier for QPACK checkpoints is 0x4B.

> *Note to readers:* This header does not currently exist in the main draft, but has manifested in several PRs, and would need to be resurrected.

Following the stream header, a checkpoint stream contains its checkpoint ID as an 8-bit prefix integer. The remainder of the stream’s data consists of the instructions defined in this section.

Data on checkpoint streams SHOULD be processed as soon as it arrives. If multiple checkpoint streams are received at once, a decoder SHOULD process data on each as it arrives if it has sent "SETTINGS_QPACK_BLOCKING_PERMITTED", but MAY process checkpoint streams one at a time.

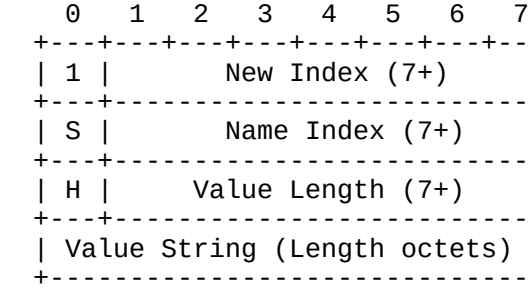
3.2.1. INSERT

An addition to the dynamic table starts with the ’1’ one-bit pattern, followed by the new index of the header represented as an integer with a 7-bit prefix. The decoder adds the supplied header to the checkpoint currently being processed, which is in the NEW state.

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the "S" bit indicates whether the reference is to the static (S=1) or dynamic (S=0) table and the index of the entry is represented as an integer with an 7-bit prefix (see Section 5.1 of [RFC7541]). This value is always non-zero.

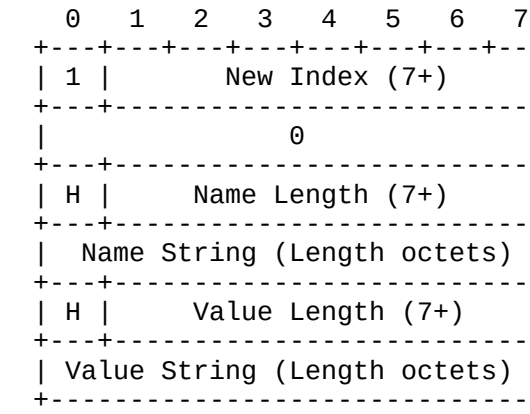
If an INSERT instruction uses an existing dynamic table entry for the name of an entry being added to the NEW checkpoint, both the existing

entry and the new entry are referenced by the NEW checkpoint. INSERT instructions which reference the dynamic table MUST reference only entries which are already included in a LIVE checkpoint. This avoids the possibility of one checkpoint stream blocking on a different checkpoint.



INSERT instruction -- Indexed Name

Otherwise, the header field name is represented as a string literal (see Section 5.2 of [RFC7541]). A value 0 is used in place of the table reference, followed by the header field name.



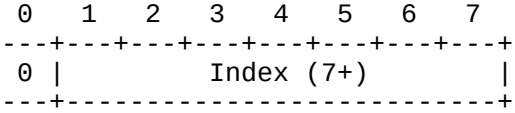
INSERT instruction -- New Name

Either form of header field name representation is followed by the header field value represented as a string literal (see Section 5.2 of [RFC7541]).

An encoder MUST NOT attempt to place a value at an index not known to be vacant. A decoder MUST treat the attempt to insert into an occupied slot or reference a name in a vacant slot as a fatal error.

3.2.2. TOUCH

This instruction is emitted to link a NEW checkpoint to an existing header table entry created by a previous checkpoint. This causes the entry not to be removed from the table so long as the current checkpoint is alive.



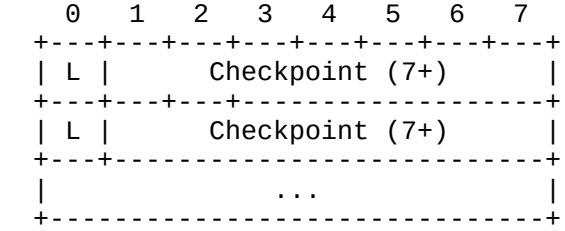
Indexed Header Field

The encoder SHOULD NOT issue multiple TOUCH commands for the same entry in the context of the same NEW checkpoint. If a non-existent index is specified, the decoder MUST treat is as an error.

3.3. Request Streams

Frames which carry HTTP message headers begin with an optional preface indicating potentially-blocking references in the frame. If present, this preface indicates that the request depends on one or more checkpoints which were NEW or PENDING for the encoder when the frame was generated. If these checkpoints are not LIVE on the decoder, it MAY delay reading the remainder of the frame until they are. (If any of these checkpoints have already been dropped, this must be treated as a stream error of type ERROR_QPACK_INVALID_REFERENCE.)

The preface is formatted as follows:

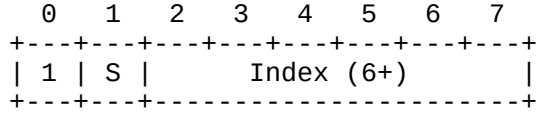


QPACK preface

The "L" bit indicates that this checkpoint is the last checkpoint in the preface; if the bit is unset (0), then another checkpoint follows.

3.3.1. Indexed Header Field Representation

An indexed header field representation identifies an entry in either the static table or the dynamic table and causes that header field to be added to the decoded header list, as described in Section 3.2 of [RFC7541].



Indexed Header Field

An indexed header field starts with the ’1’ 1-bit pattern, followed by the "S" bit indicating whether the reference is into the static (S=1) or dynamic (S=0) table. Finally, the index of the matching header field is represented as an integer with a 6-bit prefix (see Section 5.1 of [RFC7541]).

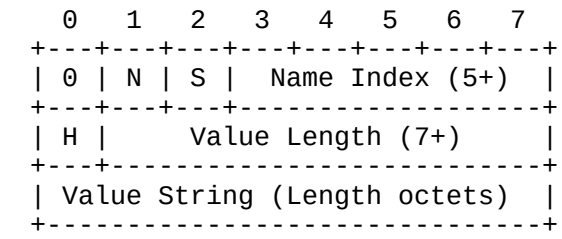
The index value of 0 is not used. It MUST be treated as a decoding error if found in an indexed header field representation.

3.3.2. Literal Header Field Representation

A literal header field representation starts with the ’0’ 1-bit pattern and causes a header field to be added the decoded header list.

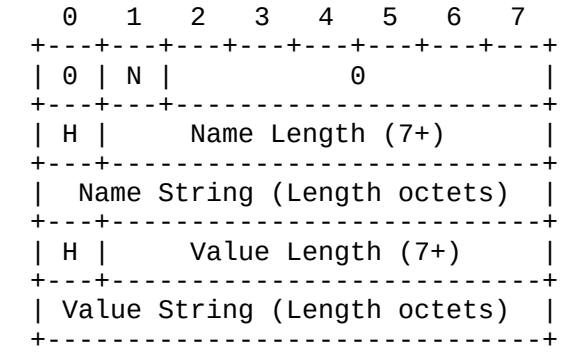
The second bit, ’N’, indicates whether an intermediary is permitted to add this header to the dynamic header table on subsequent hops. When the ’N’ bit is set, the encoded header MUST always be encoded with this specific literal representation. In particular, when a peer sends a header field that it received represented as a literal header field with the ’N’ bit set, it MUST use the same representation to forward this header field. This bit is intended for protecting header field values that are not to be put at risk by compressing them (see Section 7.1 of [RFC7541] for more details).

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the "S" bit indicates whether the reference is to the static (S=1) or dynamic (S=0) table and the index of the entry is represented as an integer with an 5-bit prefix (see Section 5.1 of [RFC7541]). This value is always non-zero.



Literal Header Field -- Indexed Name

Otherwise, the header field name is represented as a string literal (see Section 5.2 of [RFC7541]). A value 0 is used in place of the 6-bit index, followed by the header field name.



Literal Header Field -- Literal Name

Either form of header field name representation is followed by the header field value represented as a string literal (see Section 5.2 of [RFC7541]).

4. Use in HTTP/QUIC

HTTP/QUIC [I-D.ietf-quic-http] currently retains the HPACK encoder/decoder from HTTP/2, but restricts the size of the dynamic table to zero. Using QPACK instead would entail the following changes:

- Header Blocks consist of QPACK data instead of HPACK data
- HEADERS and PUSH_PROMISE frames define a flag indicating the presence of a preface.
- Just as unidirectional push streams have a stream header identifying their Push ID, a header will need to be added to differentiate checkpoint streams from pushes

- Stream 2 is reserved for the Feedback Stream

A HEADERS or PUSH_PROMISE frame MAY contain an arbitrary number of QPACK instructions. A partial HEADERS or PUSH_PROMISE frame MAY be processed upon arrival and the resulting partial header set emitted or buffered according to implementation requirements.

4.1. SETTING_QPACK_BLOCKING_PERMITTED

An HTTP/QUIC implementation can trade off the complexity of its QPACK decoder against compression efficiency by permitting the peer's compressor to reference unacknowledged entries. In the case of loss on a checkpoint stream, such references might cause the processing of request streams to block, waiting for the arrival of missing data.

If the decoder permits the encoder to make blocking references, it sets "SETTING_QPACK_BLOCKING_PERMITTED" (0xSETTING-TBD1) to a non-zero value. The encoder receiving this setting MAY encode up to this number of potentially-blocking references at a time.

Sending this setting with no value indicates that a decoder is willing to tolerate blocking references bounded only by the allowed number of streams. If a decoder does not send this setting or sends this setting with a value of zero, the encoder MUST NOT encode a header using a reference that might block.

4.2. SETTING_QPACK_INITIAL_CHECKPOINT

An HTTP/QUIC implementation MAY include the "SETTING_QPACK_INITIAL_CHECKPOINT" (0xSETTING_TBD2) setting, containing the full serialization of an initial checkpoint stream's data. If present, this setting MUST be fully processed by the peer before decoding any checkpoint streams or header frames on request streams.

The checkpoint defined by this setting is considered LIVE by both the encoder and the decoder from the beginning of the connection. The decoder does not need to send an ACK_FLUSH message confirming receipt of this setting.

5. Implementation trade-offs

This document specifies a means for the encoder to express the choices it made while encoding, but intentionally does not mandate what those choices should be. In this section, potential areas for implementation tuning are explored.

5.1. Compression Efficiency versus Blocking Avoidance

If blocking references are permitted, they will block if the frame containing the entry definition is lost or delayed. Encoders MAY choose to trade off compression efficiency and avoid blocking by using literal instructions rather than referencing the dynamic table until the insertion is believed to be complete.

The most efficient compression algorithm will reference a table entry whenever it exists in the table, but risks blocking when subject to packet loss or reordering. The most conservative algorithm will always emit literals to guarantee that no blocking will ever occur. Most implementations will choose a balance between these two extremes.

Better efficiency while being similarly conservative can be achieved by permitting references to table entries only once these entries are confirmed to be present in the table. More optimization can be achieved when the reference is known to be in the same packet as the definition.

Increases in efficiency can be achieved by assuming greater risk of blocking - implementations might choose a particular balance, or adjust their aggressiveness based on observed network characteristics.

Since it is possible to insert header values without emitting them on a stream, an encoder MAY also proactively insert header values which it believes will be needed on future requests, at the cost of reduced compression efficiency for incorrect predictions.

The ability to split updates to the header table into discrete checkpoints reduces the possibility for head-of-line blocking within the checkpoint streams. Implementations SHOULD limit the size of checkpoints to avoid head-of-line blocking within these messages.

5.2. Timely State Transitions versus Decoder Complexity

Anything which prevent checkpoints from transitioning from DYING to DEAD can prevent the encoder from adding any new entries due to the maximum table size. This does not block the encoder from continuing to make requests, but could sharply limit compression performance. Encoders would be well-served to begin moving checkpoint to DYING in advance of encountering the table maximum. Decoders SHOULD be prompt about emitting STREAM_DONE and ACK_DROP instructions to enable the encoder to recover the table space.

Similarly, for decoders which prohibit blocking references, delaying the transition of a checkpoint from PENDING to LIVE will degrade compression performance. Decoders SHOULD consume checkpoint data and emit ACK_FLUSH frames as promptly as possible.

Since decoders cannot safely drop old checkpoints until they have fully processed any checkpoints which might have been open concurrently, a long-lived checkpoint can delay the completion of an ACK_DROP. Encoders SHOULD flush all NEW checkpoints as soon as feasible after issuing a DROP instruction.

6. Security Considerations

A malicious encoder might attempt to consume a large amount of space on the decoder, but as each decoder chooses how much memory to allow the peer to consume, this state is bounded.

A malicious encoder might also send blocking references to entries which will never actually be defined. This attack is comparable to a "slow loris" attack in which a request is delivered very slowly in an attempt to consume resources on the server. Similar mitigations (request timers, etc.) SHOULD be employed to guard against such attacks.

7. IANA Considerations

This document registers two settings and one error code with the corresponding HTTP/QUIC registries.

7.1. Settings

This document registers two entries in the "HTTP/QUIC Settings" registry established by [I-D.ietf-quic-http].

Setting Name: SETTING_QPACK_BLOCKING_PERMITTED

Code: 0xSETTING-TBD1

Specification: Section 4.1

and

Setting Name: SETTING_QPACK_INITIAL_CHECKPOINT

Code: 0xSETTING-TBD2

Specification: Section 4.2

7.2. Errors

This document registers one error code in the "HTTP/QUIC Error Code" registry established by [I-D.ietf-quic-http].

Error name: ERROR_QPACK_INVALID_REFERENCE

Code: 0xERROR-TBD

Description: A blocking reference was received by a decoder which did not permit it

Specification: Section 2.2.1

8. Acknowledgements

This draft draws heavily on the text of [RFC7541], and adopts (with adaptation) the checkpoint model from [QMIN]. The direct and indirect input of those authors is gratefully acknowledged, as well as ideas gleefully stolen from:

- Jana Iyengar
- Patrick McManus
- Martin Thomson
- Charles 'Buck' Krasic
- Kyle Rose
- Alan Frindell

A substantial portion of Mike's work on this draft was supported by Microsoft during his employment there.

9. References

9.1. Normative References

[I-D.ietf-quic-http]
Bishop, M., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-07 (work in progress), October 2017.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-07 (work in progress), October 2017.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <https://www.rfc-editor.org/info/rfc7541>.

9.2. Informative References

[QMIN] Tikhonov, D., "QMIN: Header Compression for QUIC", draft-tikhonov-quic-qmin-00 (work in progress), November 2017.

Author’s Address

Mike Bishop
Akamai

Email: mbishop@evequefou.be

QUIC
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

M. Bishop, Ed.
Microsoft

March 13, 2017

Hypertext Transfer Protocol (HTTP) over QUIC
draft-ietf-quic-http-02

Abstract

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC. This document also identifies HTTP/2 features that are subsumed by QUIC, and describes how HTTP/2 extensions can be ported to QUIC.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic .

Working Group information can be found at https://github.com/quicwg ; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/http .

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 14, 2017.

Table of Contents

1. Introduction

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC, drawing heavily on the existing TCP mapping, HTTP/2. Specifically, this document identifies HTTP/2 features that are subsumed by QUIC, and describes how the other features can be implemented atop QUIC.

QUIC is described in [QUIC-TRANSPORT]. For a full description of HTTP/2, see [RFC7540].

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

2. QUIC Advertisement

An HTTP origin advertises the availability of an equivalent HTTP/QUIC endpoint via the Alt-Svc HTTP response header or the HTTP/2 ALTSVC frame ([RFC7838]), using the ALPN token defined in Section 3.

For example, an origin could indicate in an HTTP/1.1 or HTTP/2 response that HTTP/QUIC was available on UDP port 443 at the same hostname by including the following header in any response:

```
Alt-Svc: hq=":443"
```

On receipt of an Alt-Svc header indicating HTTP/QUIC support, a client MAY attempt to establish a QUIC connection to the indicated host and port and, if successful, send HTTP requests using the mapping described in this document.

Connectivity problems (e.g. firewall blocking UDP) can result in QUIC connection establishment failure, in which case the client SHOULD continue using the existing connection or try another alternative endpoint offered by the origin.

2.1. QUIC Version Hints

This document defines the "quic" parameter for Alt-Svc, which MAY be used to provide version-negotiation hints to HTTP/QUIC clients. QUIC versions are four-octet sequences with no additional constraints on format. Syntax:

```
quic = version-number
version-number = 1*8HEXDIG; hex-encoded QUIC version
```

Leading zeros SHOULD be omitted for brevity. When multiple versions are supported, the "quic" parameter MAY be repeated multiple times in a single Alt-Svc entry. For example, if a server supported both version 0x00000001 and the version rendered in ASCII as "Q034", it could specify the following header:

```
Alt-Svc: hq=":443";quic=1;quic=51303334
```

Where multiple versions are listed, the order of the values reflects the server's preference (with the first value being the most preferred version). Origins SHOULD list only versions which are supported by the alternative, but MAY omit supported versions for any reason.

3. Connection Establishment

HTTP/QUIC connections are established as described in [QUIC-TRANSPORT]. During connection establishment, HTTP/QUIC support is indicated by selecting the ALPN token "hq" in the crypto handshake.

While connection-level options pertaining to the core QUIC protocol are set in the initial crypto handshake, HTTP-specific settings are conveyed in the SETTINGS frame. After the QUIC connection is established, a SETTINGS frame (Section 5.2.3) MUST be sent as the initial frame of the HTTP control stream (StreamID 3, see Section 4). The server MUST NOT send data on any other stream until the client's SETTINGS frame has been received.

3.1. Draft Version Identification

> *RFC Editor's Note:* Please remove this section prior to publication of a final version of this document.

Only implementations of the final, published RFC can identify themselves as "hq". Until such an RFC exists, implementations MUST NOT identify themselves using this string.

Implementations of draft versions of the protocol MUST add the string "-" and the corresponding draft number to the identifier. For example, draft-ietf-quic-http-01 is identified using the string "hq-01".

Non-compatible experiments that are based on these draft versions MUST append the string "-" and an experiment name to the identifier. For example, an experimental implementation based on draft-ietf-quic-http-09 which reserves an extra stream for unsolicited transmission of 1980s pop music might identify itself as "hq-09-rickroll". Note that any label MUST conform to the "token" syntax defined in Section 3.2.6 of [RFC7230]. Experimenters are encouraged to coordinate their experiments on the quic@ietf.org mailing list.

4. Stream Mapping and Usage

A QUIC stream provides reliable in-order delivery of bytes, but makes no guarantees about order of delivery with regard to bytes on other streams. On the wire, data is framed into QUIC STREAM frames, but this framing is invisible to the HTTP framing layer. A QUIC receiver buffers and orders received STREAM frames, exposing the data contained within as a reliable byte stream to the application.

QUIC reserves Stream 1 for crypto operations (the handshake, crypto config updates). Stream 3 is reserved for sending and receiving HTTP control frames, and is analogous to HTTP/2's Stream 0. This connection control stream is considered critical to the HTTP connection. If the connection control stream is closed for any reason, this MUST be treated as a connection error of type QUIC_CLOSED_CRITICAL_STREAM.

When HTTP headers and data are sent over QUIC, the QUIC layer handles most of the stream management. An HTTP request/response consumes a pair of streams: This means that the client's first request occurs on QUIC streams 5 and 7, the second on stream 9 and 11, and so on. The server's first push consumes streams 2 and 4. This amounts to the second least-significant bit differentiating the two streams in a request.

The lower-numbered stream is called the message control stream and carries frames related to the request/response, including HEADERS. The higher-numbered stream is the data stream and carries the request/response body with no additional framing. Note that a request or response without a body will cause this stream to be half-closed in the corresponding direction without transferring data.

Because the message control stream contains HPACK data which manipulates connection-level state, the message control stream MUST NOT be closed with a stream-level error. If an implementation chooses to reject a request with a QUIC error code, it MUST trigger a QUIC RST_STREAM on the data stream only. An implementation MAY close (FIN) a message control stream without completing a full HTTP message if the data stream has been abruptly closed. Data on message control streams MUST be fully consumed, or the connection terminated.

All message control streams are considered critical to the HTTP connection. If a message control stream is terminated abruptly for any reason, this MUST be treated as a connection error of type HTTP_RST_CONTROL_STREAM. When a message control stream terminates cleanly, if the last frame on the stream was truncated, this MUST be treated as a connection error (see HTTP_MALFORMED_* in Section 6.1).

Pairs of streams must be utilized sequentially, with no gaps. The data stream is opened at the same time as the message control stream is opened and is closed after transferring the body. The data stream is closed immediately after sending the request headers if there is no body.

HTTP does not need to do any separate multiplexing when using QUIC - data sent over a QUIC stream always maps to a particular HTTP transaction. Requests and responses are considered complete when the corresponding QUIC streams are closed in the appropriate direction.

4.1. Stream 3: Connection Control Stream

Since most connection-level concerns will be managed by QUIC, the primary use of Stream 3 will be for the SETTINGS frame when the connection opens and for PRIORITY frames subsequently.

4.2. HTTP Message Exchanges

A client sends an HTTP request on a new pair of QUIC streams. A server sends an HTTP response on the same streams as the request.

An HTTP message (request or response) consists of:

1. one header block (see Section 5.2.1) on the control stream containing the message headers (see [RFC7230], Section 3.2),

2. the payload body (see [RFC7230], Section 3.3), sent on the data stream,

3. optionally, one header block on the control stream containing the trailer-part, if present (see [RFC7230], Section 4.1.2).

In addition, prior to sending the message header block indicated above, a response may contain zero or more header blocks on the control stream containing the message headers of informational (1xx) HTTP responses (see [RFC7230], Section 3.2 and [RFC7231], Section 6.2).

The data stream MUST be half-closed immediately after the transfer of the body. If the message does not contain a body, the corresponding data stream MUST still be half-closed without transferring any data. The "chunked" transfer encoding defined in Section 4.1 of [RFC7230] MUST NOT be used.

Trailing header fields are carried in an additional header block on the message control stream. Such a header block is a sequence of HEADERS frames with End Header Block set on the last frame. Senders MUST send only one header block in the trailers section; receivers MUST decode any subsequent header blocks in order to maintain HPACK decoder state, but the resulting output MUST be discarded.

An HTTP request/response exchange fully consumes a pair of streams. After sending a request, a client closes the streams for sending; after sending a response, the server closes its streams for sending and the QUIC streams are fully closed.

A server can send a complete response prior to the client sending an entire request if the response does not depend on any portion of the request that has not been sent and received. When this is true, a server MAY request that the client abort transmission of a request without error by sending a RST_STREAM with an error code of NO_ERROR after sending a complete response and closing its stream. Clients MUST NOT discard responses as a result of receiving such a RST_STREAM, though clients can always discard responses at their discretion for other reasons.

4.2.1. Header Compression

HTTP/QUIC uses HPACK header compression as described in [RFC7541]. HPACK was designed for HTTP/2 with the assumption of in-order delivery such as that provided by TCP. A sequence of encoded header

blocks must arrive (and be decoded) at an endpoint in the same order in which they were encoded. This ensures that the dynamic state at the two endpoints remains in sync.

QUIC streams provide in-order delivery of data sent on those streams, but there are no guarantees about order of delivery between streams. To achieve in-order delivery of HEADERS frames in QUIC, the HPACK-bearing frames contain a counter which can be used to ensure in-order processing. Data (request/response bodies) which arrive out of order are buffered until the corresponding HEADERS arrive.

This does introduce head-of-line blocking: if the packet containing HEADERS for stream N is lost or reordered then the HEADERS for stream N+4 cannot be processed until it has been retransmitted successfully, even though the HEADERS for stream N+4 may have arrived.

DISCUSS: Keep HPACK with HOLB? Redesign HPACK to be order-invariant? How much do we need to retain compatibility with HTTP/2's HPACK?

4.2.2. The CONNECT Method

The pseudo-method CONNECT ([RFC7231], Section 4.3.6) is primarily used with HTTP proxies to establish a TLS session with an origin server for the purposes of interacting with "https" resources. In HTTP/1.x, CONNECT is used to convert an entire HTTP connection into a tunnel to a remote host. In HTTP/2, the CONNECT method is used to establish a tunnel over a single HTTP/2 stream to a remote host for similar purposes.

A CONNECT request in HTTP/QUIC functions in the same manner as in HTTP/2. The request MUST be formatted as described in [RFC7540], Section 8.3. A CONNECT request that does not conform to these restrictions is malformed. The message data stream MUST NOT be closed at the end of the request.

A proxy that supports CONNECT establishes a TCP connection ([RFC0793]) to the server identified in the ":authority" pseudo-header field. Once this connection is successfully established, the proxy sends a HEADERS frame containing a 2xx series status code to the client, as defined in [RFC7231], Section 4.3.6, on the message control stream.

All QUIC STREAM frames on the message data stream correspond to data sent on the TCP connection. Any QUIC STREAM frame sent by the client is transmitted by the proxy to the TCP server; data received from the TCP server is written to the data stream by the proxy. Note that the

size and number of TCP segments is not guaranteed to map predictably to the size and number of QUIC STREAM frames.

The TCP connection can be closed by either peer. When the client half-closes the data stream, the proxy will set the FIN bit on its connection to the TCP server. When the proxy receives a packet with the FIN bit set, it will half-close the corresponding data stream. TCP connections which remain half-closed in a single direction are not invalid, but are often handled poorly by servers, so clients SHOULD NOT half-close connections on which they are still expecting data.

A TCP connection error is signaled with RST_STREAM. A proxy treats any error in the TCP connection, which includes receiving a TCP segment with the RST bit set, as a stream error of type HTTP_CONNECT_ERROR (Section 6.1). Correspondingly, a proxy MUST send a TCP segment with the RST bit set if it detects an error with the stream or the QUIC connection.

4.3. Stream Priorities

HTTP/QUIC uses the priority scheme described in [RFC7540] Section 5.3. In this priority scheme, a given stream can be designated as dependent upon another stream, which expresses the preference that the latter stream (the "parent" stream) be allocated resources before the former stream (the "dependent" stream). Taken together, the dependencies across all streams in a connection form a dependency tree. The structure of the dependency tree changes as PRIORITY frames add, remove, or change the dependency links between streams.

For consistency’s sake, all PRIORITY frames MUST refer to the message control stream of the dependent request, not the data stream.

4.4. Server Push

HTTP/QUIC supports server push as described in [RFC7540]. During connection establishment, the client indicates whether it is willing to receive server pushes via the SETTINGS_DISABLE_PUSH setting in the SETTINGS frame (see Section 3), which defaults to 1 (true).

As with server push for HTTP/2, the server initiates a server push by sending a PUSH_PROMISE frame containing the StreamID of the stream to be pushed, as well as request header fields attributed to the request. The PUSH_PROMISE frame is sent on the control stream of the associated (client-initiated) request, while the Promised Stream ID field specifies the Stream ID of the control stream for the server-initiated request.

The server push response is conveyed in the same way as a non-server-push response, with response headers and (if present) trailers carried by HEADERS frames sent on the control stream, and response body (if any) sent via the corresponding data stream.

5. HTTP Framing Layer

Frames are used only on the connection (stream 3) and message (streams 5, 9, etc.) control streams. Other streams carry data payload and are not framed at the HTTP layer.

This section describes HTTP framing in QUIC and highlights some differences from HTTP/2 framing. For more detail on differences from HTTP/2, see Section 7.1.

5.1. Frame Layout

All frames have the following format:

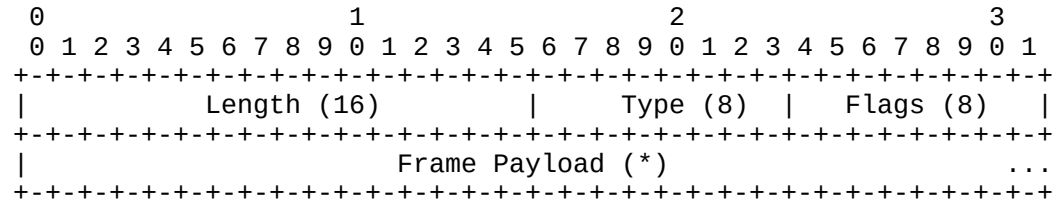


Figure 1: HTTP/QUIC frame format

5.2. Frame Definitions

5.2.1. HEADERS

The HEADERS frame (type=0x1) is used to carry part of a header set, compressed using HPACK [RFC7541].

One flag is defined:

End Header Block (0x4): This frame concludes a header block.

A HEADERS frame with any other flags set MUST be treated as a connection error of type HTTP_MALFORMED_HEADERS.

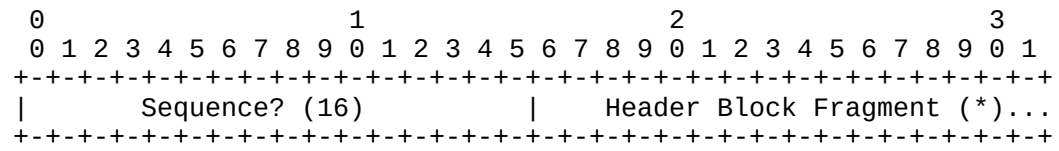


Figure 2: HEADERS frame payload

The HEADERS frame payload has the following fields:

Sequence Number: Present only on the first frame of a header block sequence. This MUST be set to zero on the first header block sequence, and incremented on each header block.

The next frame on the same stream after a HEADERS frame without the EHB flag set MUST be another HEADERS frame. A receiver MUST treat the receipt of any other type of frame as a stream error of type HTTP_INTERRUPTED_HEADERS. (Note that QUIC can intersperse data from other streams between frames, or even during transmission of frames, so multiplexing is not blocked by this requirement.)

A full header block is contained in a sequence of zero or more HEADERS frames without EHB set, followed by a HEADERS frame with EHB set.

On receipt, header blocks (HEADERS, PUSH_PROMISE) MUST be processed by the HPACK decoder in sequence. If a block is missing, all subsequent HPACK frames MUST be held until it arrives, or the connection terminated.

When the Sequence counter reaches its maximum value (0xFFFF), the next increment returns it to zero. An endpoint MUST NOT wrap the Sequence counter to zero until the previous zero-value header block has been confirmed received.

5.2.2. PRIORITY

The PRIORITY (type=0x02) frame specifies the sender-advised priority of a stream and is substantially different from [RFC7540]. In order to support ordering, it MUST be sent only on the connection control stream. The format has been modified to accommodate not being sent on-stream and the larger stream ID space of QUIC.

The semantics of the Stream Dependency, Weight, and E flag are the same as in HTTP/2.

The flags defined are:

E (0x01): Indicates that the stream dependency is exclusive (see [RFC7540] Section 5.3).

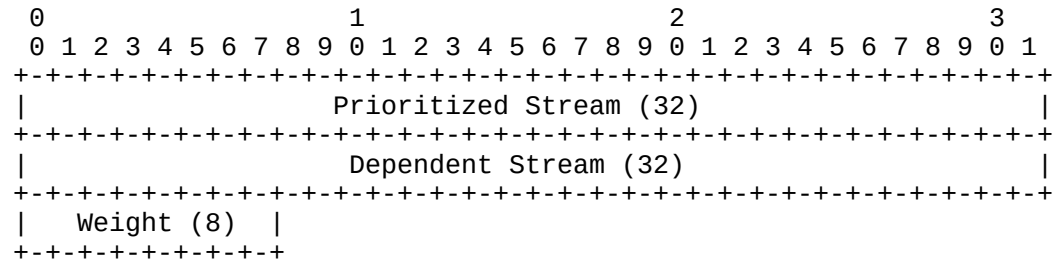


Figure 3: PRIORITY frame payload

The HEADERS frame payload has the following fields:

Prioritized Stream: A 32-bit stream identifier for the message control stream whose priority is being updated.

Stream Dependency: A 32-bit stream identifier for the stream that this stream depends on (see Section 4.3 and {!RFC7540} Section 5.3).

Weight: An unsigned 8-bit integer representing a priority weight for the stream (see [RFC7540] Section 5.3). Add one to the value to obtain a weight between 1 and 256.

A PRIORITY frame MUST have a payload length of nine octets. A PRIORITY frame of any other length MUST be treated as a connection error of type HTTP_MALFORMED_PRIORITY.

5.2.3. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate, such as preferences and constraints on peer behavior, and is substantially different from [RFC7540]. Individually, a SETTINGS parameter can also be referred to as a "setting".

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer, which can be used by the receiving peer. However, a negotiation can be implied by the use of SETTINGS - a peer uses SETTINGS to advertise a set of supported values. The recipient can then choose which entries from this list are also acceptable and proceed with the value it has chosen. (This choice could be announced in a field of an extension frame, or in its own value in SETTINGS.)

Different values for the same parameter can be advertised by each peer. For example, a client might permit a very large HPACK state table while a server chooses to use a small one to conserve memory.

Parameters MUST NOT occur more than once. A receiver MAY treat the presence of the same parameter more than once as a connection error of type HTTP_MALFORMED_SETTINGS.

The SETTINGS frame defines no flags.

The payload of a SETTINGS frame consists of zero or more parameters, each consisting of an unsigned 16-bit setting identifier and a length-prefixed binary value.

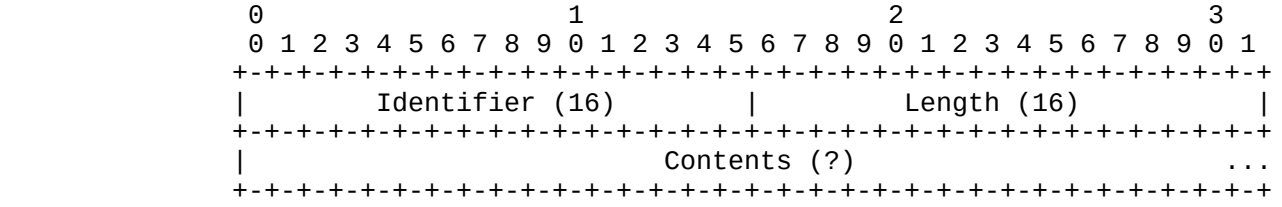


Figure 4: SETTINGS value format

A zero-length content indicates that the setting value is a Boolean and true. False is indicated by the absence of the setting.

Non-zero-length values MUST be compared against the remaining length of the SETTINGS frame. Any value which purports to cross the end of the frame MUST cause the SETTINGS frame to be considered malformed and trigger a connection error of type HTTP_MALFORMED_SETTINGS.

An implementation MUST ignore the contents for any SETTINGS identifier it does not understand.

SETTINGS frames always apply to a connection, never a single stream. A SETTINGS frame MUST be sent as the first frame of the connection control stream (see Section 4) by each peer, and MUST NOT be sent subsequently or on any other stream. If an endpoint receives an SETTINGS frame on a different stream, the endpoint MUST respond with a connection error of type HTTP_SETTINGS_ON_WRONG_STREAM. If an endpoint receives a second SETTINGS frame, the endpoint MUST respond with a connection error of type HTTP_MULTIPLE_SETTINGS.

The SETTINGS frame affects connection state. A badly formed or incomplete SETTINGS frame MUST be treated as a connection error (Section 5.4.1) of type HTTP_MALFORMED_SETTINGS.

5.2.3.1. Integer encoding

Settings which are integers are transmitted in network byte order. Leading zero octets are permitted, but implementations SHOULD use only as many bytes as are needed to represent the value. An integer MUST NOT be represented in more bytes than would be used to transfer the maximum permitted value.

5.2.3.2. Defined SETTINGS Parameters

The following settings are defined in HTTP/QUIC:

SETTINGS_HEADER_TABLE_SIZE (0x1): An integer with a maximum value of 2^32 - 1.

SETTINGS_DISABLE_PUSH (0x2): Transmitted as a Boolean; replaces SETTINGS_ENABLE_PUSH

SETTINGS_MAX_HEADER_LIST_SIZE (0x6): An integer with a maximum value of 2^32 - 1.

5.2.3.3. Usage in 0-RTT

When a 0-RTT QUIC connection is being used, the client’s initial requests will be sent before the arrival of the server’s SETTINGS frame. Clients SHOULD cache at least the following settings about servers:

- SETTINGS_HEADER_TABLE_SIZE
- SETTINGS_MAX_HEADER_LIST_SIZE

Clients MUST comply with cached settings until the server’s current settings are received. If a client does not have cached values, it SHOULD assume the following values:

- SETTINGS_HEADER_TABLE_SIZE: 0 octets
- SETTINGS_MAX_HEADER_LIST_SIZE: 16,384 octets

Servers MAY continue processing data from clients which exceed its current configuration during the initial flight. In this case, the client MUST apply the new settings immediately upon receipt.

If the connection is closed because these or other constraints were violated during the 0-RTT flight (e.g. with HTTP_HPACK_DECOMPRESSION_FAILED), clients MAY establish a new connection and retry any 0-RTT requests using the settings sent by

the server on the closed connection. (This assumes that only requests that are safe to retry are sent in 0-RTT.) If the connection was closed before the SETTINGS frame was received, clients SHOULD discard any cached values and use the defaults above on the next connection.

5.2.4. PUSH_PROMISE

The PUSH_PROMISE frame (type=0x05) is used to carry a request header set from server to client, as in HTTP/2. It defines no flags.

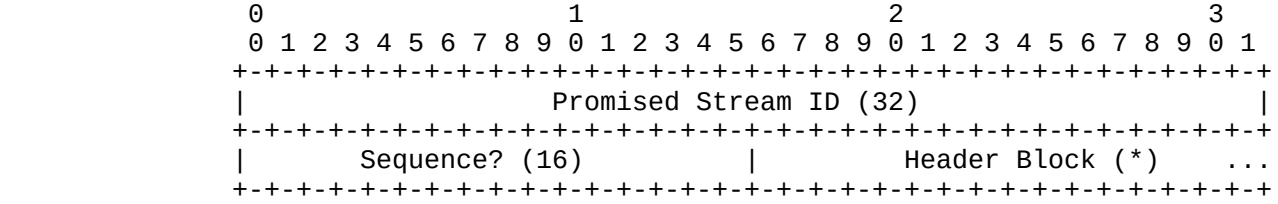


Figure 5: PUSH_PROMISE frame payload

The payload consists of:

Promised Stream ID: A 32-bit Stream ID indicating the QUIC stream on which the response headers will be sent. (The response body stream is implied by the headers stream, as defined in Section 4.)

HPACK Sequence: A sixteen-bit counter, equivalent to the Sequence field in HEADERS

Payload: HPACK-compressed request headers for the promised response.

6. Error Handling

QUIC allows the application to abruptly terminate individual streams or the entire connection when an error is encountered. These are referred to as "stream errors" or "connection errors" and are described in more detail in [QUIC-TRANSPORT].

HTTP/QUIC requires that only data streams be terminated abruptly. Terminating a message control stream will result in an error of type HTTP_RST_CONTROL_STREAM.

This section describes HTTP-specific error codes which can be used to express the cause of a connection or stream error.

6.1. HTTP-Defined QUIC Error Codes

QUIC allocates error codes 0x0000-0x3FFF to application protocol definition. The following error codes are defined by HTTP for use in QUIC RST_STREAM, GOAWAY, and CONNECTION_CLOSE frames.

HTTP_PUSH_REFUSED (0x01): The server has attempted to push content which the client will not accept on this connection.

HTTP_INTERNAL_ERROR (0x02): An internal error has occurred in the HTTP stack.

HTTP_PUSH_ALREADY_IN_CACHE (0x03): The server has attempted to push content which the client has cached.

HTTP_REQUEST_CANCELLED (0x04): The client no longer needs the requested data.

HTTP_HPACK_DECOMPRESSION_FAILED (0x05): HPACK failed to decompress a frame and cannot continue.

HTTP_CONNECT_ERROR (0x06): The connection established in response to a CONNECT request was reset or abnormally closed.

HTTP_EXCESSIVE_LOAD (0x07): The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

HTTP_VERSION_FALLBACK (0x08): The requested operation cannot be served over HTTP/QUIC. The peer should retry over HTTP/2.

HTTP_MALFORMED_HEADERS (0x09): A HEADERS frame has been received with an invalid format.

HTTP_MALFORMED_PRIORITY (0x0A): A PRIORITY frame has been received with an invalid format.

HTTP_MALFORMED_SETTINGS (0x0B): A SETTINGS frame has been received with an invalid format.

HTTP_MALFORMED_PUSH_PROMISE (0x0C): A PUSH_PROMISE frame has been received with an invalid format.

HTTP_INTERRUPTED_HEADERS (0x0E): A HEADERS frame without the End Header Block flag was followed by a frame other than HEADERS.

HTTP_SETTINGS_ON_WRONG_STREAM (0x0F): A SETTINGS frame was received on a request control stream.

HTTP_MULTIPLE_SETTINGS (0x10): More than one SETTINGS frame was received.

HTTP_RST_CONTROL_STREAM (0x11): A message control stream closed abruptly.

7. Considerations for Transitioning from HTTP/2

HTTP/QUIC is strongly informed by HTTP/2, and bears many similarities. This section points out important differences from HTTP/2 and describes how to map HTTP/2 extensions into HTTP/QUIC.

7.1. HTTP Frame Types

Many framing concepts from HTTP/2 can be elided away on QUIC, because the transport deals with them. Because frames are already on a stream, they can omit the stream number. Because frames do not block multiplexing (QUIC's multiplexing occurs below this layer), the support for variable-maximum-length packets can be removed. Because stream termination is handled by QUIC, an END_STREAM flag is not required.

Frame payloads are largely drawn from [RFC7540]. However, QUIC includes many features (e.g. flow control) which are also present in HTTP/2. In these cases, the HTTP mapping does not re-implement them. As a result, several HTTP/2 frame types are not required in HTTP/QUIC. Where an HTTP/2-defined frame is no longer used, the frame ID has been reserved in order to maximize portability between HTTP/2 and HTTP/QUIC implementations. However, even equivalent frames between the two mappings are not identical.

Many of the differences arise from the fact that HTTP/2 provides an absolute ordering between frames across all streams, while QUIC provides this guarantee on each stream only. As a result, if a frame type makes assumptions that frames from different streams will still be received in the order sent, HTTP/QUIC will break them.

For example, implicit in the HTTP/2 prioritization scheme is the notion of in-order delivery of priority changes (i.e., dependency tree mutations): since operations on the dependency tree such as reparenting a subtree are not commutative, both sender and receiver must apply them in the same order to ensure that both sides have a consistent view of the stream dependency tree. HTTP/2 specifies priority assignments in PRIORITY frames and (optionally) in HEADERS frames. To achieve in-order delivery of priority changes in HTTP/QUIC, PRIORITY frames are sent on the connection control stream and the PRIORITY section is removed from the HEADERS frame.

Other than this issue, frame type HTTP/2 extensions are typically portable to QUIC simply by replacing Stream 0 in HTTP/2 with Stream 3 in HTTP/QUIC.

Below is a listing of how each HTTP/2 frame type is mapped:

DATA (0x0): Instead of DATA frames, HTTP/QUIC uses a separate data stream. See Section 4.

HEADERS (0x1): As described above, the PRIORITY region of HEADERS is not supported. A separate PRIORITY frame MUST be used. Padding is not defined in HTTP/QUIC frames. See Section 5.2.1.

PRIORITY (0x2): As described above, the PRIORITY frame is sent on the connection control stream. See Section 5.2.2.

RST_STREAM (0x3): RST_STREAM frames do not exist, since QUIC provides stream lifecycle management.

SETTINGS (0x4): SETTINGS frames are sent only at the beginning of the connection. See Section 5.2.3 and Section 7.2.

PUSH_PROMISE (0x5): See Section 5.2.4.

PING (0x6): PING frames do not exist, since QUIC provides equivalent functionality.

GOAWAY (0x7): GOAWAY frames do not exist, since QUIC provides equivalent functionality.

WINDOW_UPDATE (0x8): WINDOW_UPDATE frames do not exist, since QUIC provides flow control.

CONTINUATION (0x9): CONTINUATION frames do not exist; instead, larger HEADERS/PUSH_PROMISE frames than HTTP/2 are permitted, and HEADERS frames can be used in series.

The IANA registry of frame types has been updated in Section 9.3 to include references to the definition for each frame type in HTTP/2 and in HTTP/QUIC. Frames not defined as available in HTTP/QUIC SHOULD NOT be sent and SHOULD be ignored as unknown on receipt.

7.2. HTTP/2 SETTINGS Parameters

An important difference from HTTP/2 is that settings are sent once, at the beginning of the connection, and thereafter cannot change. This eliminates many corner cases around synchronization of changes.

Some transport-level options that HTTP/2 specifies via the SETTINGS frame are superseded by QUIC transport parameters in HTTP/QUIC. The HTTP-level options that are retained in HTTP/QUIC have the same value as in HTTP/2.

Below is a listing of how each HTTP/2 SETTINGS parameter is mapped:

SETTINGS_HEADER_TABLE_SIZE: See Section 5.2.3.2.

SETTINGS_ENABLE_PUSH: See SETTINGS_DISABLE_PUSH in Section 5.2.3.2.

SETTINGS_MAX_CONCURRENT_STREAMS: QUIC requires the maximum number of incoming streams per connection to be specified in the initial transport handshake. Specifying SETTINGS_MAX_CONCURRENT_STREAMS in the SETTINGS frame is an error.

SETTINGS_INITIAL_WINDOW_SIZE: QUIC requires both stream and connection flow control window sizes to be specified in the initial transport handshake. Specifying SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS frame is an error.

SETTINGS_MAX_FRAME_SIZE: This setting has no equivalent in HTTP/QUIC. Specifying it in the SETTINGS frame is an error.

SETTINGS_MAX_HEADER_LIST_SIZE: See Section 5.2.3.2.

Settings defined by extensions to HTTP/2 MAY be expressed as integers with a maximum value of 2^32-1, if they are applicable to HTTP/QUIC, but SHOULD have a specification describing their usage. Fields for this purpose have been added to the IANA registry in Section 9.4.

7.3. HTTP/2 Error Codes

QUIC has the same concepts of "stream" and "connection" errors that HTTP/2 provides. However, because the error code space is shared between multiple components, there is no direct portability of HTTP/2 error codes.

The HTTP/2 error codes defined in Section 7 of [RFC7540] map to QUIC error codes as follows:

NO_ERROR (0x0): QUIC_NO_ERROR

PROTOCOL_ERROR (0x1): No single mapping. See new HTTP_MALFORMED_* error codes defined in Section 6.1.

INTERNAL_ERROR (0x2) HTTP_INTERNAL_ERROR in Section 6.1.

FLOW_CONTROL_ERROR (0x3): Not applicable, since QUIC handles flow control. Would provoke a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA from the QUIC layer.

SETTINGS_TIMEOUT (0x4): Not applicable, since no acknowledgement of SETTINGS is defined.

STREAM_CLOSED (0x5): Not applicable, since QUIC handles stream management. Would provoke a QUIC_STREAM_DATA_AFTER_TERMINATION from the QUIC layer.

FRAME_SIZE_ERROR (0x6) No single mapping. See new error codes defined in Section 6.1.

REFUSED_STREAM (0x7): Not applicable, since QUIC handles stream management. Would provoke a QUIC_TOO_MANY_OPEN_STREAMS from the QUIC layer.

CANCEL (0x8): HTTP_REQUEST_CANCELLED in Section 6.1.

COMPRESSION_ERROR (0x9): HTTP_HPACK_DECOMPRESSION_FAILED in Section 6.1.

CONNECT_ERROR (0xa): HTTP_CONNECT_ERROR in Section 6.1.

ENHANCE_YOUR_CALM (0xb): HTTP_EXCESSIVE_LOAD in Section 6.1.

INADEQUATE_SECURITY (0xc): Not applicable, since QUIC is assumed to provide sufficient security on all connections.

HTTP_1_1_REQUIRED (0xd): HTTP_VERSION_FALLBACK in Section 6.1.

Error codes defined by HTTP/2 extensions need to be re-registered for HTTP/QUIC if still applicable. See Section 9.5.

8. Security Considerations

The security considerations of HTTP over QUIC should be comparable to those of HTTP/2.

The modified SETTINGS format contains nested length elements, which could pose a security risk to an uncautious implementer. A SETTINGS frame parser MUST ensure that the length of the frame exactly matches the length of the settings it contains.

9. IANA Considerations

9.1. Registration of HTTP/QUIC Identification String

This document creates a new registration for the identification of HTTP/QUIC in the "Application Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [RFC7301].

The "hq" string identifies HTTP/QUIC:

Protocol: HTTP over QUIC

Identification Sequence: 0x68 0x71 ("hq")

Specification: This document

9.2. Registration of QUIC Version Hint Alt-Svc Parameter

This document creates a new registration for version-negotiation hints in the "Hypertext Transfer Protocol (HTTP) Alt-Svc Parameter" registry established in [RFC7838].

Parameter: "quic"

Specification: This document, Section 2.1

9.3. Existing Frame Types

This document adds two new columns to the "HTTP/2 Frame Type" registry defined in [RFC7540]:

Supported Protocols: Indicates which associated protocols use the frame type. Values MUST be one of:

* "HTTP/2 only"

* "HTTP/QUIC only"

* "Both"

HTTP/QUIC Specification: Indicates where this frame’s behavior over QUIC is defined; required if the frame is supported over QUIC.

Values for existing registrations are assigned by this document:

Frame Type	Supported Protocols	HTTP/QUIC Specification
DATA	HTTP/2 only	N/A
HEADERS	Both	Section 5.2.1
PRIORITY	Both	Section 5.2.2
RST_STREAM	HTTP/2 only	N/A
SETTINGS	Both	Section 5.2.3
PUSH_PROMISE	Both	Section 5.2.4
PING	HTTP/2 only	N/A
GOAWAY	HTTP/2 only	N/A
WINDOW_UPDATE	HTTP/2 only	N/A
CONTINUATION	HTTP/2 only	N/A

The "Specification" column is renamed to "HTTP/2 specification" and is only required if the frame is supported over HTTP/2.

9.4. Settings Parameters

This document adds two new columns to the "HTTP/2 Settings" registry defined in [RFC7540]:

Supported Protocols: Indicates which associated protocols use the setting. Values MUST be one of:

* "HTTP/2 only"

* "HTTP/QUIC only"

* "Both"

HTTP/QUIC Specification: Indicates where this setting’s behavior over QUIC is defined; required if the frame is supported over QUIC.

Values for existing registrations are assigned by this document:

Setting Name	Supported Protocols	HTTP/QUIC Specification
HEADER_TABLE_SIZE	Both	Section 5.2.3.2
ENABLE_PUSH / DISABLE_PUSH	Both	Section 5.2.3.2
MAX_CONCURRENT_STREAMS	HTTP/2 Only	N/A
INITIAL_WINDOW_SIZE	HTTP/2 Only	N/A
MAX_FRAME_SIZE	HTTP/2 Only	N/A
MAX_HEADER_LIST_SIZE	Both	Section 5.2.3.2

The "Specification" column is renamed to "HTTP/2 Specification" and is only required if the setting is supported over HTTP/2.

9.5. Error Codes

This document establishes a registry for HTTP/QUIC error codes. The "HTTP/QUIC Error Code" registry manages a 30-bit space. The "HTTP/QUIC Error Code" registry operates under the "Expert Review" policy [RFC5226].

Registrations for error codes are required to include a description of the error code. An expert reviewer is advised to examine new registrations for possible duplication with existing error codes. Use of existing registrations is to be encouraged, but not mandated.

New registrations are advised to provide the following information:

Name: A name for the error code. Specifying an error code name is optional.

Code: The 30-bit error code value.

Description: A brief description of the error code semantics, longer if no detailed specification is provided.

Specification: An optional reference for a specification that defines the error code.

The entries in the following table are registered by this document.

Name	Code	Description	Specification
HTTP_PUSH_REFUSED	0x01	Client refused pushed content	Section 6.1
HTTP_INTERNAL_ERROR	0x02	Internal error	Section 6.1
HTTP_PUSH_ALREADY_IN_CACHE	0x03	Pushed content already cached	Section 6.1
HTTP_REQUEST_CANCELLED	0x04	Data no longer needed	Section 6.1
HTTP_HPACK_DECOMPRESSION_FAILED	0x05	HPACK cannot continue	Section 6.1
HTTP_CONNECT_ERROR	0x06	TCP reset or error on CONNECT request	Section 6.1
HTTP_EXCESSIVE_LOAD	0x07	Peer generating excessive load	Section 6.1
HTTP_VERSION_FALLBACK	0x08	Retry over HTTP/2	Section 6.1
HTTP_MALFORMED_HEADERS	0x09	Invalid HEADERS frame	Section 6.1
HTTP_MALFORMED_PRIORITY	0x0A	Invalid PRIORITY frame	Section 6.1
HTTP_MALFORMED_SETTINGS	0x0B	Invalid SETTINGS frame	Section 6.1

HTTP_MALFORMED_PUSH_PROMISE	0x0C	Invalid PUSH_PROMISE frame	Section 6.1
HTTP_INTERRUPTED_HEADERS	0x0E	Incomplete HEADERS block	Section 6.1
HTTP_SETTINGS_ON_WRONG_STREAM	0x0F	SETTINGS frame on a request control stream	Section 6.1
HTTP_MULTIPLE_SETTINGS	0x10	Multiple SETTINGS frames	Section 6.1
HTTP_RST_CONTROL_STREAM	0x11	Message control stream was RST	Section 6.1

10. References

10.1. Normative References

[QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport".

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <http://www.rfc-editor.org/info/rfc793>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <http://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <http://www.rfc-editor.org/info/rfc7231>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <http://www.rfc-editor.org/info/rfc7540>.

[RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <http://www.rfc-editor.org/info/rfc7541>.

[RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <http://www.rfc-editor.org/info/rfc7838>.

10.2. Informative References

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <http://www.rfc-editor.org/info/rfc7301>.

Appendix A. Contributors

The original authors of this specification were Robbie Shade and Mike Warres.

Appendix B. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

B.1. Since draft-ietf-quic-http-01:

- SETTINGS changes (#181):
  - SETTINGS can be sent only once at the start of a connection; no changes thereafter
  - SETTINGS_ACK removed

    * Settings can only occur in the SETTINGS frame a single time

    * Boolean format updated

  - Alt-Svc parameter changed from "v" to "quic"; format updated (#229)

  - Closing the connection control stream or any message control stream is a fatal error (#176)

  - HPACK Sequence counter can wrap (#173)

  - 0-RTT guidance added

  - Guide to differences from HTTP/2 and porting HTTP/2 extensions added (#127,#242)

B.2. Since draft-ietf-quic-http-00:

  - Changed "HTTP/2-over-QUIC" to "HTTP/QUIC" throughout (#11,#29)

  - Changed from using HTTP/2 framing within Stream 3 to new framing format and two-stream-per-request model (#71,#72,#73)

  - Adopted SETTINGS format from draft-bishop-httpbis-extended-settings-01

  - Reworked SETTINGS_ACK to account for indeterminate inter-stream order (#75)

  - Described CONNECT pseudo-method (#95)

  - Updated ALPN token and Alt-Svc guidance (#13,#87)

  - Application-layer-defined error codes (#19,#74)

B.3. Since draft-shade-quic-http2-mapping-00:

  - Adopted as base for draft-ietf-quic-http.

  - Updated authors/editors list.

Author’s Address

Mike Bishop (editor)
Microsoft

Email: Michael.Bishop@microsoft.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: October 19, 2018

M. Bishop, Ed.
Akamai

April 17, 2018

Hypertext Transfer Protocol (HTTP) over QUIC
draft-ietf-quic-http-11

Abstract

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC. This document also identifies HTTP/2 features that are subsumed by QUIC, and describes how HTTP/2 extensions can be ported to QUIC.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic [1].

Working Group information can be found at https://github.com/quicwg [2]; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/-http [3].

Status of This Memo

Table of Contents

1. Introduction

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC, drawing heavily on the existing TCP mapping, HTTP/2. Specifically, this document identifies HTTP/2 features that are subsumed by QUIC, and describes how the other features can be implemented atop QUIC.

QUIC is described in [QUIC-TRANSPORT]. For a full description of HTTP/2, see [RFC7540].

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Field definitions are given in Augmented Backus-Naur Form (ABNF), as defined in [RFC5234].

This document uses the variable-length integer encoding from [QUIC-TRANSPORT].

Protocol elements called "frames" exist in both this document and [QUIC-TRANSPORT]. Where frames from [QUIC-TRANSPORT] are referenced, the frame name will be prefaced with "QUIC." For example, "QUIC APPLICATION_CLOSE frames." References without this preface refer to frames defined in Section 4.2.

2. Connection Setup and Management

2.1. Discovering an HTTP/QUIC Endpoint

An HTTP origin advertises the availability of an equivalent HTTP/QUIC endpoint via the Alt-Svc HTTP response header or the HTTP/2 ALTSVC frame ([RFC7838]), using the ALPN token defined in Section 2.2.

For example, an origin could indicate in an HTTP/1.1 or HTTP/2 response that HTTP/QUIC was available on UDP port 50781 at the same hostname by including the following header in any response:

```
Alt-Svc: hq=":50781"
```

On receipt of an Alt-Svc record indicating HTTP/QUIC support, a client MAY attempt to establish a QUIC connection to the indicated host and port and, if successful, send HTTP requests using the mapping described in this document.

Connectivity problems (e.g. firewall blocking UDP) can result in QUIC connection establishment failure, in which case the client SHOULD continue using the existing connection or try another alternative endpoint offered by the origin.

Servers MAY serve HTTP/QUIC on any UDP port, since an alternative always includes an explicit port.

2.1.1. QUIC Version Hints

This document defines the "quic" parameter for Alt-Svc, which MAY be used to provide version-negotiation hints to HTTP/QUIC clients. QUIC versions are four-octet sequences with no additional constraints on format. Leading zeros SHOULD be omitted for brevity.

Syntax:

```
quic = DQUOTE version-number [ "," version-number ] * DQUOTE
version-number = 1*8HEXDIG; hex-encoded QUIC version
```

Where multiple versions are listed, the order of the values reflects the server's preference (with the first value being the most preferred version). Reserved versions MAY be listed, but unreserved versions which are not supported by the alternative SHOULD NOT be present in the list. Origins MAY omit supported versions for any reason.

Clients MUST ignore any included versions which they do not support. The "quic" parameter MUST NOT occur more than once; clients SHOULD process only the first occurrence.

For example, suppose a server supported both version 0x00000001 and the version rendered in ASCII as "Q034". If it opted to include the reserved versions (from Section 4 of [QUIC-TRANSPORT]) 0x0 and 0x1abadaba, it could specify the following header:

```
Alt-Svc: hq=":49288";quic="1,1abadaba,51303334,0"
```

A client acting on this header would drop the reserved versions (because it does not support them), then attempt to connect to the alternative using the first version in the list which it does support.

2.2. Connection Establishment

HTTP/QUIC relies on QUIC as the underlying transport. The QUIC version being used MUST use TLS version 1.3 or greater as its handshake protocol. The Server Name Indication (SNI) extension [RFC6066] MUST be included in the TLS handshake.

QUIC connections are established as described in [QUIC-TRANSPORT]. During connection establishment, HTTP/QUIC support is indicated by selecting the ALPN token "hq" in the TLS handshake. Support for other application-layer protocols MAY be offered in the same handshake.

While connection-level options pertaining to the core QUIC protocol are set in the initial crypto handshake, HTTP-specific settings are conveyed in the SETTINGS frame. After the QUIC connection is established, a SETTINGS frame (Section 4.2.5) MUST be sent by each endpoint as the initial frame of their respective HTTP control stream (Stream ID 2 or 3, see Section 3). The server MUST NOT send data on any other stream until the client's SETTINGS frame has been received.

2.2.1. Draft Version Identification

> *RFC Editor's Note:* Please remove this section prior to publication of a final version of this document.

Only implementations of the final, published RFC can identify themselves as "hq". Until such an RFC exists, implementations MUST NOT identify themselves using this string.

Implementations of draft versions of the protocol MUST add the string "-" and the corresponding draft number to the identifier. For example, draft-ietf-quic-http-01 is identified using the string "hq-01".

Non-compatible experiments that are based on these draft versions MUST append the string "-" and an experiment name to the identifier. For example, an experimental implementation based on draft-ietf-quic-http-09 which reserves an extra stream for unsolicited transmission of 1980s pop music might identify itself as "hq-09-rickroll". Note that any label MUST conform to the "token" syntax defined in Section 3.2.6 of [RFC7230]. Experimenters are encouraged to coordinate their experiments on the quic@ietf.org mailing list.

2.3. Connection Reuse

Once a connection exists to a server endpoint, this connection MAY be reused for requests with multiple different URI authority components. The client MAY send any requests for which the client considers the server authoritative.

An authoritative HTTP/QUIC endpoint is typically discovered because the client has received an Alt-Svc record from the request's origin which nominates the endpoint as a valid HTTP Alternative Service for that origin. As required by [RFC7838], clients MUST check that the nominated server can present a valid certificate for the origin before considering it authoritative. Clients MUST NOT assume that an HTTP/QUIC endpoint is authoritative for other origins without an explicit signal.

A server that does not wish clients to reuse connections for a particular origin can indicate that it is not authoritative for a request by sending a 421 (Misdirected Request) status code in response to the request (see Section 9.1.2 of [RFC7540]).

3. Stream Mapping and Usage

A QUIC stream provides reliable in-order delivery of bytes, but makes no guarantees about order of delivery with regard to bytes on other streams. On the wire, data is framed into QUIC STREAM frames, but this framing is invisible to the HTTP framing layer. A QUIC receiver buffers and orders received STREAM frames, exposing the data contained within as a reliable byte stream to the application.

QUIC reserves the first client-initiated, bidirectional stream (Stream 0) for cryptographic operations. HTTP over QUIC reserves the first unidirectional stream sent by either peer (Streams 2 and 3) for sending and receiving HTTP control frames. This pair of unidirectional streams is analogous to HTTP/2's Stream 0. The data sent on these streams is critical to the HTTP connection. If either control stream is closed for any reason, this MUST be treated as a connection error of type QUIC_CLOSED_CRITICAL_STREAM.

When HTTP headers and data are sent over QUIC, the QUIC layer handles most of the stream management.

An HTTP request/response consumes a single client-initiated, bidirectional stream. A bidirectional stream ensures that the response can be readily correlated with the request. This means that the client's first request occurs on QUIC stream 4, with subsequent requests on stream 8, 12, and so on.

Server push uses server-initiated, unidirectional streams. Thus, the server's first push consumes stream 7 and subsequent pushes use stream 11, 15, and so on.

These streams carry frames related to the request/response (see Section 4.2). When a stream terminates cleanly, if the last frame on the stream was truncated, this MUST be treated as a connection error (see HTTP_MALFORMED_FRAME in Section 6.1). Streams which terminate abruptly may be reset at any point in the frame.

Streams SHOULD be used sequentially, with no gaps.

HTTP does not need to do any separate multiplexing when using QUIC - data sent over a QUIC stream always maps to a particular HTTP transaction. Requests and responses are considered complete when the corresponding QUIC stream is closed in the appropriate direction.

3.1. Control Streams

Since most connection-level concerns will be managed by QUIC, the primary use of Streams 2 and 3 will be for the SETTINGS frame when the connection opens and for PRIORITY frames subsequently.

A pair of unidirectional streams is used rather than a single bidirectional stream. This allows either peer to send data as soon they are able. Depending on whether 0-RTT is enabled on the connection, either client or server might be able to send stream data first after the cryptographic handshake completes.

3.2. HTTP Message Exchanges

A client sends an HTTP request on a client-initiated, bidirectional QUIC stream. A server sends an HTTP response on the same stream as the request.

An HTTP message (request or response) consists of:

1. one header block (see Section 4.2.2) containing the message headers (see [RFC7230], Section 3.2),

2. the payload body (see [RFC7230], Section 3.3), sent as a series of DATA frames (see Section 4.2.1),

3. optionally, one header block containing the trailer-part, if present (see [RFC7230], Section 4.1.2).

In addition, prior to sending the message header block indicated above, a response may contain zero or more header blocks containing the message headers of informational (1xx) HTTP responses (see [RFC7230], Section 3.2 and [RFC7231], Section 6.2).

PUSH_PROMISE frames MAY be interleaved with the frames of a response message indicating a pushed resource related to the response. These PUSH_PROMISE frames are not part of the response, but carry the headers of a separate HTTP request message. See Section 3.4 for more details.

The "chunked" transfer encoding defined in Section 4.1 of [RFC7230] MUST NOT be used.

Trailing header fields are carried in an additional header block following the body. Such a header block is a sequence of HEADERS frames with End Header Block set on the last frame. Senders MUST send only one header block in the trailers section; receivers MUST discard any subsequent header blocks.

An HTTP request/response exchange fully consumes a QUIC stream. After sending a request, a client closes the stream for sending; after sending a response, the server closes the stream for sending and the QUIC stream is fully closed.

A server can send a complete response prior to the client sending an entire request if the response does not depend on any portion of the request that has not been sent and received. When this is true, a server MAY request that the client abort transmission of a request without error by triggering a QUIC STOP_SENDING with error code HTTP_EARLY_RESPONSE, sending a complete response, and cleanly closing its streams. Clients MUST NOT discard complete responses as a result of having their request terminated abruptly, though clients can always discard responses at their discretion for other reasons. Servers MUST NOT abort a response in progress as a result of receiving a solicited RST_STREAM.

3.2.1. Header Compression

HTTP/QUIC uses QPACK header compression as described in [QPACK], a variation of HPACK which allows the flexibility to avoid header-compression-induced head-of-line blocking. See that document for additional details.

3.2.2. The CONNECT Method

The pseudo-method CONNECT ([RFC7231], Section 4.3.6) is primarily used with HTTP proxies to establish a TLS session with an origin server for the purposes of interacting with "https" resources. In HTTP/1.x, CONNECT is used to convert an entire HTTP connection into a tunnel to a remote host. In HTTP/2, the CONNECT method is used to establish a tunnel over a single HTTP/2 stream to a remote host for similar purposes.

A CONNECT request in HTTP/QUIC functions in the same manner as in HTTP/2. The request MUST be formatted as described in [RFC7540], Section 8.3. A CONNECT request that does not conform to these restrictions is malformed. The request stream MUST NOT be half-closed at the end of the request.

A proxy that supports CONNECT establishes a TCP connection ([RFC0793]) to the server identified in the ":authority" pseudo-header field. Once this connection is successfully established, the proxy sends a HEADERS frame containing a 2xx series status code to the client, as defined in [RFC7231], Section 4.3.6.

All DATA frames on the request stream correspond to data sent on the TCP connection. Any DATA frame sent by the client is transmitted by

the proxy to the TCP server; data received from the TCP server is packaged into DATA frames by the proxy. Note that the size and number of TCP segments is not guaranteed to map predictably to the size and number of HTTP DATA or QUIC STREAM frames.

The TCP connection can be closed by either peer. When the client ends the request stream (that is, the receive stream at the proxy enters the "Data Recvd" state), the proxy will set the FIN bit on its connection to the TCP server. When the proxy receives a packet with the FIN bit set, it will terminate the send stream that it sends to client. TCP connections which remain half-closed in a single direction are not invalid, but are often handled poorly by servers, so clients SHOULD NOT cause send a STREAM frame with a FIN bit for connections on which they are still expecting data.

A TCP connection error is signaled with RST_STREAM. A proxy treats any error in the TCP connection, which includes receiving a TCP segment with the RST bit set, as a stream error of type HTTP_CONNECT_ERROR (Section 6.1). Correspondingly, a proxy MUST send a TCP segment with the RST bit set if it detects an error with the stream or the QUIC connection.

3.2.3. Request Cancellation

Either client or server can cancel requests by closing the stream (QUIC RST_STREAM or STOP_SENDING frames, as appropriate) with an error type of HTTP_REQUEST_CANCELLED (Section 6.1). When the client cancels a request or response, it indicates that the response is no longer of interest.

When the server cancels either direction of the request stream using HTTP_REQUEST_CANCELLED, it indicates that no application processing was performed. The client can treat requests cancelled by the server as though they had never been sent at all, thereby allowing them to be retried later on a new connection. Servers MUST NOT use the HTTP_REQUEST_CANCELLED status for requests which were partially or fully processed.

Note: In this context, "processed" means that some data from the stream was passed to some higher layer of software that might have taken some action as a result.

If a stream is cancelled after receiving a complete response, the client MAY ignore the cancellation and use the response. However, if a stream is cancelled after receiving a partial response, the response SHOULD NOT be used. Automatically retrying such requests is not possible, unless this is otherwise permitted (e.g., idempotent actions like GET, PUT, or DELETE).

3.3. Request Prioritization

HTTP/QUIC uses the priority scheme described in [RFC7540], Section 5.3. In this priority scheme, a given request can be designated as dependent upon another request, which expresses the preference that the latter stream (the "parent" request) be allocated resources before the former stream (the "dependent" request). Taken together, the dependencies across all requests in a connection form a dependency tree. The structure of the dependency tree changes as PRIORITY frames add, remove, or change the dependency links between requests.

The PRIORITY frame Section 4.2.3 identifies a request either by identifying the stream that carries a request or by using a Push ID (Section 4.2.6).

Only a client can send PRIORITY frames. A server MUST NOT send a PRIORITY frame.

3.4. Server Push

HTTP/QUIC supports server push in a similar manner to [RFC7540], but uses different mechanisms. During connection establishment, the client enables server push by sending a MAX_PUSH_ID frame (see Section 4.2.9). A server cannot use server push until it receives a MAX_PUSH_ID frame.

As with server push for HTTP/2, the server initiates a server push by sending a PUSH_PROMISE frame (see Section 4.2.6) that includes request headers for the promised request. Promised requests MUST conform to the requirements in Section 8.2 of [RFC7540].

The PUSH_PROMISE frame is sent on the client-initiated, bidirectional stream that carried the request that generated the push. This allows the server push to be associated with a request. Ordering of a PUSH_PROMISE in relation to certain parts of the response is important (see Section 8.2.1 of [RFC7540]).

Unlike HTTP/2, the PUSH_PROMISE does not reference a stream; it contains a Push ID. The Push ID uniquely identifies a server push. This allows a server to fulfill promises in the order that best suits its needs.

When a server later fulfills a promise, the server push response is conveyed on a push stream. A push stream is a server-initiated, unidirectional stream. A push stream identifies the Push ID of the promise that it fulfills, encoded as a variable-length integer.

A server SHOULD use Push IDs sequentially, starting at 0. A client uses the MAX_PUSH_ID frame (Section 4.2.9) to limit the number of pushes that a server can promise. A client MUST treat receipt of a push stream with a Push ID that is greater than the maximum Push ID as a connection error of type HTTP_PUSH_LIMIT_EXCEEDED.

If a promised server push is not needed by the client, the client SHOULD send a CANCEL_PUSH frame; if the push stream is already open, a QUIC STOP_SENDING frame with an appropriate error code can be used instead (e.g., HTTP_PUSH_REFUSED, HTTP_PUSH_ALREADY_IN_CACHE; see Section 6). This asks the server not to transfer the data and indicates that it will be discarded upon receipt.

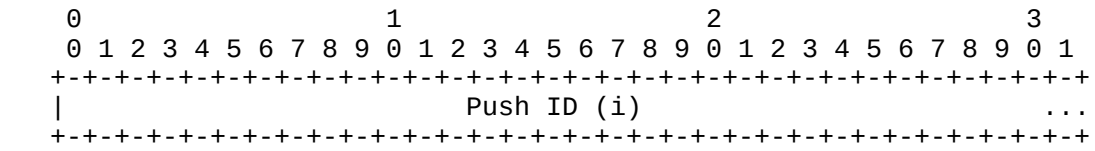


Figure 1: Push Stream Header

Push streams always begin with a header containing the Push ID. Each Push ID MUST only be used once in a push stream header. If a push stream header includes a Push ID that was used in another push stream header, the client MUST treat this as a connection error of type HTTP_DUPLICATE_PUSH. The same Push ID can be used in multiple PUSH_PROMISE frames (see Section 4.2.6).

After the header, a push stream contains a response (Section 3.2), with response headers, a response body (if any) carried by DATA frames, then trailers (if any) carried by HEADERS frames.

4. HTTP Framing Layer

Frames are used on each stream. This section describes HTTP framing in QUIC and highlights some differences from HTTP/2 framing. For more detail on differences from HTTP/2, see Section 7.2.

4.1. Frame Layout

All frames have the following format:

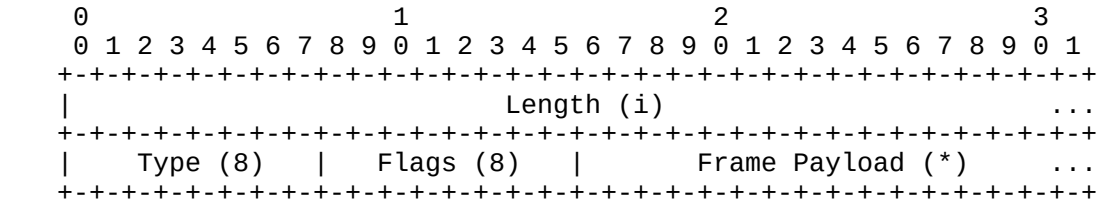


Figure 2: HTTP/QUIC frame format

A frame includes the following fields:

Length: A variable-length integer that describes the length of the Frame Payload. This length does not include the frame header.

Type: An 8-bit type for the frame.

Flags: An 8-bit field containing flags. The Type field determines the semantics of flags.

Frame Payload: A payload, the semantics of which are determined by the Type field.

4.2. Frame Definitions

4.2.1. DATA

DATA frames (type=0x0) convey arbitrary, variable-length sequences of octets associated with an HTTP request or response payload.

The DATA frame defines no flags.

DATA frames MUST be associated with an HTTP request or response. If a DATA frame is received on either control stream, the recipient MUST respond with a connection error (Section 6) of type HTTP_WRONG_STREAM.

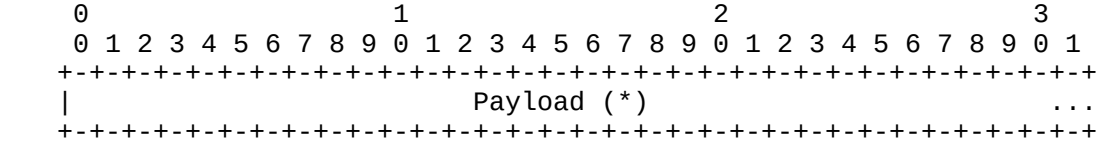


Figure 3: DATA frame payload

DATA frames MUST contain a non-zero-length payload. If a DATA frame is received with a payload length of zero, the recipient MUST respond with a stream error (Section 6) of type HTTP_MALFORMED_FRAME.

4.2.2. HEADERS

The HEADERS frame (type=0x1) is used to carry a header block, compressed using QPACK. See [QPACK] for more details.

The HEADERS frame defines a single flag:

BLOCKING (0x01): Indicates the stream might need to wait for dependent headers before processing. If 0, the frame can be processed immediately upon receipt.

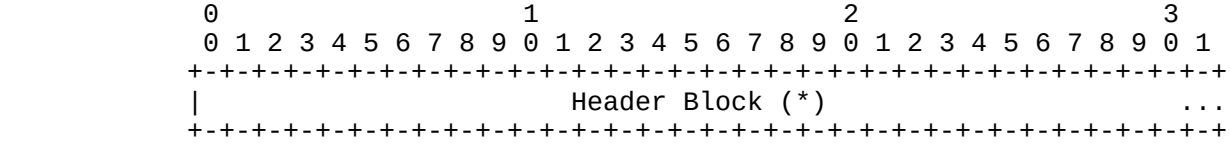


Figure 4: HEADERS frame payload

HEADERS frames can be sent on the Control Stream as well as on request / push streams. The value of BLOCKING MUST be 0 for HEADERS frames on the Control Stream, since they can only depend on previous HEADERS on the same stream.

4.2.3. PRIORITY

The PRIORITY (type=0x02) frame specifies the sender-advised priority of a stream and is substantially different in format from [RFC7540]. In order to ensure that prioritization is processed in a consistent order, PRIORITY frames MUST be sent on the control stream. A PRIORITY frame sent on any other stream MUST be treated as a HTTP_WRONG_STREAM error.

The format has been modified to accommodate not being sent on a request stream, to allow for identification of server pushes, and the larger stream ID space of QUIC. The semantics of the Stream Dependency, Weight, and E flag are otherwise the same as in HTTP/2.

The flags defined are:

PUSH_PRIORITIZED (0x04): Indicates that the Prioritized Stream is a server push rather than a request.

PUSH_DEPENDENT (0x02): Indicates a dependency on a server push.

E (0x01): Indicates that the stream dependency is exclusive (see [RFC7540], Section 5.3).

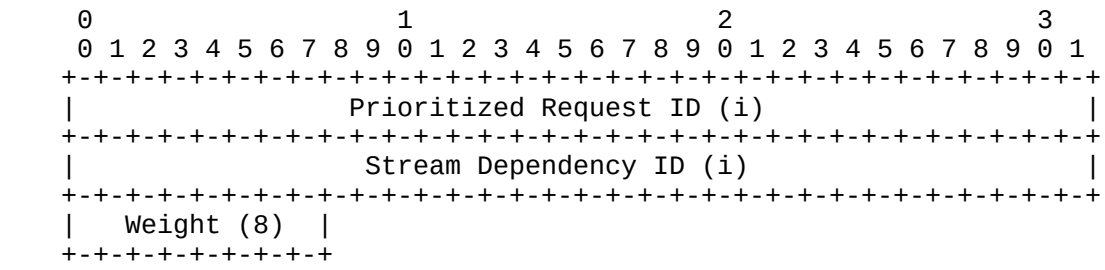


Figure 5: PRIORITY frame payload

The PRIORITY frame payload has the following fields:

Prioritized Request ID: A variable-length integer that identifies a request. This contains the Stream ID of a request stream when the PUSH_PRIORITIZED flag is clear, or a Push ID when the PUSH_PRIORITIZED flag is set.

Stream Dependency ID: A variable-length integer that identifies a dependent request. This contains the Stream ID of a request stream when the PUSH_DEPENDENT flag is clear, or a Push ID when the PUSH_DEPENDENT flag is set. A request Stream ID of 0 indicates a dependency on the root stream. For details of dependencies, see Section 3.3 and [RFC7540], Section 5.3.

Weight: An unsigned 8-bit integer representing a priority weight for the stream (see [RFC7540], Section 5.3). Add one to the value to obtain a weight between 1 and 256.

A PRIORITY frame identifies a request to prioritize, and a request upon which that request is dependent. A Prioritized Request ID or Stream Dependency ID identifies a client-initiated request using the corresponding stream ID when the corresponding PUSH_PRIORITIZED or PUSH_DEPENDENT flag is not set. Setting the PUSH_PRIORITIZED or PUSH_DEPENDENT flag causes the Prioritized Request ID or Stream Dependency ID (respectively) to identify a server push using a Push ID (see Section 4.2.6 for details).

A PRIORITY frame MAY identify a Stream Dependency ID using a Stream ID of 0; as in [RFC7540], this makes the request dependent on the root of the dependency tree.

A PRIORITY frame MUST identify a client-initiated, bidirectional stream. A server MUST treat receipt of PRIORITY frame with a Stream ID of any other type as a connection error of type HTTP_MALFORMED_FRAME.

Stream ID 0 cannot be reprioritized. A Prioritized Request ID that identifies Stream 0 MUST be treated as a connection error of type HTTP_MALFORMED_FRAME.

A PRIORITY frame that does not reference a request MUST be treated as a HTTP_MALFORMED_FRAME error, unless it references Stream ID 0. A PRIORITY that sets a PUSH_PRIORITIZED or PUSH_DEPENDENT flag, but then references a non-existent Push ID MUST be treated as a HTTP_MALFORMED_FRAME error.

A PRIORITY frame MUST contain only the identified fields. A PRIORITY frame that contains more or fewer fields, or a PRIORITY frame that includes a truncated integer encoding MUST be treated as a connection error of type HTTP_MALFORMED_FRAME.

4.2.4. CANCEL_PUSH

The CANCEL_PUSH frame (type=0x3) is used to request cancellation of server push prior to the push stream being created. The CANCEL_PUSH frame identifies a server push request by Push ID (see Section 4.2.6) using a variable-length integer.

When a server receives this frame, it aborts sending the response for the identified server push. If the server has not yet started to send the server push, it can use the receipt of a CANCEL_PUSH frame to avoid opening a stream. If the push stream has been opened by the server, the server SHOULD sent a QUIC RST_STREAM frame on those streams and cease transmission of the response.

A server can send this frame to indicate that it won't be sending a response prior to creation of a push stream. Once the push stream has been created, sending CANCEL_PUSH has no effect on the state of the push stream. A QUIC RST_STREAM frame SHOULD be used instead to cancel transmission of the server push response.

A CANCEL_PUSH frame is sent on the control stream. Sending a CANCEL_PUSH frame on a stream other than the control stream MUST be treated as a stream error of type HTTP_WRONG_STREAM.

The CANCEL_PUSH frame has no defined flags.

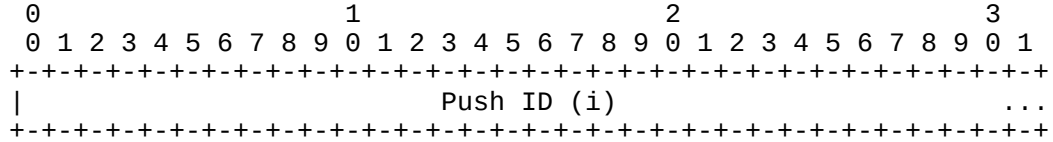


Figure 6: CANCEL_PUSH frame payload

The CANCEL_PUSH frame carries a Push ID encoded as a variable-length integer. The Push ID identifies the server push that is being cancelled (see Section 4.2.6).

If the client receives a CANCEL_PUSH frame, that frame might identify a Push ID that has not yet been mentioned by a PUSH_PROMISE frame.

An endpoint MUST treat a CANCEL_PUSH frame which does not contain exactly one properly-formatted variable-length integer as a connection error of type HTTP_MALFORMED_FRAME.

4.2.5. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate, such as preferences and constraints on peer behavior, and is different from [RFC7540]. Individually, a SETTINGS parameter can also be referred to as a "setting".

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer, which can be used by the receiving peer. However, a negotiation can be implied by the use of SETTINGS - a peer uses SETTINGS to advertise a set of supported values. The recipient can then choose which entries from this list are also acceptable and proceed with the value it has chosen. (This choice could be announced in a field of an extension frame, or in its own value in SETTINGS.)

Different values for the same parameter can be advertised by each peer. For example, a client might be willing to consume very large response headers, while servers are more cautious about request size.

Parameters MUST NOT occur more than once. A receiver MAY treat the presence of the same parameter more than once as a connection error of type HTTP_MALFORMED_FRAME.

The SETTINGS frame defines no flags.

The payload of a SETTINGS frame consists of zero or more parameters, each consisting of an unsigned 16-bit setting identifier and a length-prefixed binary value.

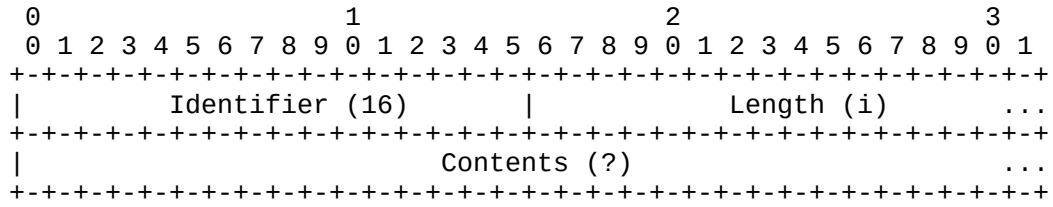


Figure 7: SETTINGS value format

A zero-length content indicates that the setting value is a Boolean and true. False is indicated by the absence of the setting.

Non-zero-length values MUST be compared against the remaining length of the SETTINGS frame. Any value which purports to cross the end of the frame MUST cause the SETTINGS frame to be considered malformed and trigger a connection error of type HTTP_MALFORMED_FRAME.

An implementation MUST ignore the contents for any SETTINGS identifier it does not understand.

SETTINGS frames always apply to a connection, never a single stream. A SETTINGS frame MUST be sent as the first frame of either control stream (see Section 3) by each peer, and MUST NOT be sent subsequently or on any other stream. If an endpoint receives an SETTINGS frame on a different stream, the endpoint MUST respond with a connection error of type HTTP_WRONG_STREAM. If an endpoint receives a second SETTINGS frame, the endpoint MUST respond with a connection error of type HTTP_MALFORMED_FRAME.

The SETTINGS frame affects connection state. A badly formed or incomplete SETTINGS frame MUST be treated as a connection error (Section 6) of type HTTP_MALFORMED_FRAME.

4.2.5.1. Integer encoding

Settings which are integers use the QUIC variable-length integer encoding.

4.2.5.2. Defined SETTINGS Parameters

The following settings are defined in HTTP/QUIC:

SETTINGS_HEADER_TABLE_SIZE (0x1): An integer with a maximum value of 2^30 - 1.

SETTINGS_MAX_HEADER_LIST_SIZE (0x6): An integer with a maximum value of 2^30 - 1

4.2.5.3. Initial SETTINGS Values

When a 0-RTT QUIC connection is being used, the client's initial requests will be sent before the arrival of the server's SETTINGS frame. Clients MUST store the settings the server provided in the session being resumed and MUST comply with stored settings until the server's current settings are received.

Servers MAY continue processing data from clients which exceed its current configuration during the initial flight. In this case, the client MUST apply the new settings immediately upon receipt.

When a 1-RTT QUIC connection is being used, the client MUST NOT send requests prior to receiving and processing the server's SETTINGS frame.

4.2.6. PUSH_PROMISE

The PUSH_PROMISE frame (type=0x05) is used to carry a request header set from server to client, as in HTTP/2. The PUSH_PROMISE frame defines a single flag:

BLOCKING (0x01): Indicates the stream might need to wait for dependent headers before processing. If 0, the frame can be processed immediately upon receipt.

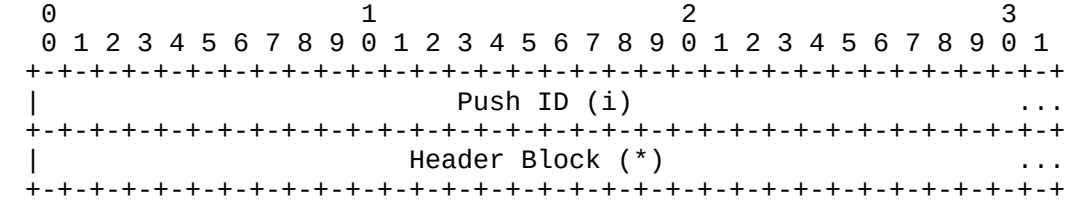


Figure 8: PUSH_PROMISE frame payload

The payload consists of:

Push ID: A variable-length integer that identifies the server push request. A push ID is used in push stream header (Section 3.4), CANCEL_PUSH frames (Section 4.2.4), and PRIORITY frames (Section 4.2.3).

Header Block: QPACK-compressed request headers for the promised response. See [QPACK] for more details.

A server MUST NOT use a Push ID that is larger than the client has provided in a MAX_PUSH_ID frame (Section 4.2.9). A client MUST treat

receipt of a PUSH_PROMISE that contains a larger Push ID than the client has advertised as a connection error of type HTTP_MALFORMED_FRAME.

A server MAY use the same Push ID in multiple PUSH_PROMISE frames. This allows the server to use the same server push in response to multiple concurrent requests. Referencing the same server push ensures that a PUSH_PROMISE can be made in relation to every response in which server push might be needed without duplicating pushes.

A server that uses the same Push ID in multiple PUSH_PROMISE frames MUST include the same header fields each time. The octets of the header block MAY be different due to differing encoding, but the header fields and their values MUST be identical. Note that ordering of header fields is significant. A client MUST treat receipt of a PUSH_PROMISE with conflicting header field values for the same Push ID as a connection error of type HTTP_MALFORMED_FRAME.

Allowing duplicate references to the same Push ID is primarily to reduce duplication caused by concurrent requests. A server SHOULD avoid reusing a Push ID over a long period. Clients are likely to consume server push responses and not retain them for reuse over time. Clients that see a PUSH_PROMISE that uses a Push ID that they have since consumed and discarded are forced to ignore the PUSH_PROMISE.

4.2.7. GOAWAY

The GOAWAY frame (type=0x7) is used to initiate graceful shutdown of a connection by a server. GOAWAY allows a server to stop accepting new requests while still finishing processing of previously received requests. This enables administrative actions, like server maintenance. GOAWAY by itself does not close a connection.

The GOAWAY frame does not define any flags.

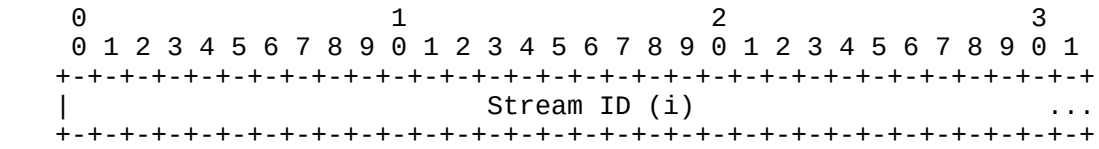


Figure 9: GOAWAY frame payload

The GOAWAY frame carries a QUIC Stream ID for a client-initiated, bidirectional stream encoded as a variable-length integer. A client MUST treat receipt of a GOAWAY frame containing a Stream ID of any other type as a connection error of type HTTP_MALFORMED_FRAME.

Clients do not need to send GOAWAY to initiate a graceful shutdown; they simply stop making new requests. A server MUST treat receipt of a GOAWAY frame as a connection error (Section 6) of type HTTP_UNEXPECTED_GOAWAY.

The GOAWAY frame applies to the connection, not a specific stream. An endpoint MUST treat a GOAWAY frame on a stream other than the control stream as a connection error (Section 6) of type HTTP_WRONG_STREAM.

New client requests might already have been sent before the client receives the server's GOAWAY frame. The GOAWAY frame contains the Stream ID of the last client-initiated request that was or might be processed in this connection, which enables client and server to agree on which requests were accepted prior to the connection shutdown. This identifier MAY be lower than the stream limit identified by a QUIC MAX_STREAM_ID frame, and MAY be zero if no requests were processed. Servers SHOULD NOT increase the MAX_STREAM_ID limit after sending a GOAWAY frame.

Once sent, the server MUST cancel requests sent on streams with an identifier higher than the included last Stream ID. Clients MUST NOT send new requests on the connection after receiving GOAWAY, although requests might already be in transit. A new connection can be established for new requests.

If the client has sent requests on streams with a higher Stream ID than indicated in the GOAWAY frame, those requests are considered cancelled (Section 3.2.3). Clients SHOULD reset any streams above this ID with the error code HTTP_REQUEST_CANCELLED. Servers MAY also cancel requests on streams below the indicated ID if these requests were not processed.

Requests on Stream IDs less than or equal to the Stream ID in the GOAWAY frame might have been processed; their status cannot be known until they are completed successfully, reset individually, or the connection terminates.

Servers SHOULD send a GOAWAY frame when the closing of a connection is known in advance, even if the advance notice is small, so that the remote peer can know whether a stream has been partially processed or not. For example, if an HTTP client sends a POST at the same time that a server closes a QUIC connection, the client cannot know if the server started to process that POST request if the server does not send a GOAWAY frame to indicate what streams it might have acted on.

For unexpected closures caused by error conditions, a QUIC CONNECTION_CLOSE or APPLICATION_CLOSE frame MUST be used. However, a

GOAWAY MAY be sent first to provide additional detail to clients and to allow the client to retry requests. Including the GOAWAY frame in the same packet as the QUIC CONNECTION_CLOSE or APPLICATION_CLOSE frame improves the chances of the frame being received by clients.

If a connection terminates without a GOAWAY frame, the last Stream ID is effectively the highest possible Stream ID (as determined by QUIC's MAX_STREAM_ID).

An endpoint MAY send multiple GOAWAY frames if circumstances change. For instance, an endpoint that sends GOAWAY without an error code during graceful shutdown could subsequently encounter an error condition. The last stream identifier from the last GOAWAY frame received indicates which streams could have been acted upon. A server MUST NOT increase the value they send in the last Stream ID, since clients might already have retried unprocessed requests on another connection.

A client that is unable to retry requests loses all requests that are in flight when the server closes the connection. A server that is attempting to gracefully shut down a connection SHOULD send an initial GOAWAY frame with the last Stream ID set to the current value of QUIC's MAX_STREAM_ID and SHOULD NOT increase the MAX_STREAM_ID thereafter. This signals to the client that a shutdown is imminent and that initiating further requests is prohibited. After allowing time for any in-flight requests (at least one round-trip time), the server MAY send another GOAWAY frame with an updated last Stream ID. This ensures that a connection can be cleanly shut down without losing requests.

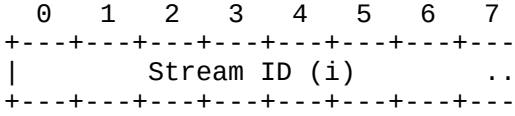
Once all requests on streams at or below the identified stream number have been completed or cancelled, and all promised server push responses associated with those requests have been completed or cancelled, the connection can be closed using an Immediate Close (see [QUIC-TRANSPORT]). An endpoint that completes a graceful shutdown SHOULD use the QUIC APPLICATION_CLOSE frame with the HTTP_NO_ERROR code.

4.2.8. HEADER_ACK

The HEADER_ACK frame (type=0x8) is used by header compression to ensure consistency. The frames are sent from the QPACK decoder to the QPACK encoder; that is, the server sends them to the client to acknowledge processing of the client's header blocks, and the client sends them to the server to acknowledge processing of the server's header blocks.

The HEADER_ACK frame is sent on the Control Stream when the QPACK decoder has fully processed a header block. It is used by the peer's QPACK encoder to determine whether subsequent indexed representations that might reference that block are vulnerable to head-of-line blocking, and to prevent eviction races. See [QPACK] for more details on the use of this information.

The HEADER_ACK frame indicates the stream on which the header block was processed by encoding the Stream ID as a variable-length integer. The same Stream ID can be identified multiple times, as multiple header-containing blocks can be sent on a single stream in the case of intermediate responses, trailers, pushed requests, etc. as well as on the Control Streams. Since header frames on each stream are received and processed in order, this gives the encoder precise feedback on which header blocks within a stream have been fully processed.



HEADER_ACK frame

The HEADER_ACK frame does not define any flags.

4.2.9. MAX_PUSH_ID

The MAX_PUSH_ID frame (type=0xD) is used by clients to control the number of server pushes that the server can initiate. This sets the maximum value for a Push ID that the server can use in a PUSH_PROMISE frame. Consequently, this also limits the number of push streams that the server can initiate in addition to the limit set by the QUIC MAX_STREAM_ID frame.

The MAX_PUSH_ID frame is always sent on a control stream. Receipt of a MAX_PUSH_ID frame on any other stream MUST be treated as a connection error of type HTTP_WRONG_STREAM.

A server MUST NOT send a MAX_PUSH_ID frame. A client MUST treat the receipt of a MAX_PUSH_ID frame as a connection error of type HTTP_MALFORMED_FRAME.

The maximum Push ID is unset when a connection is created, meaning that a server cannot push until it receives a MAX_PUSH_ID frame. A client that wishes to manage the number of promised server pushes can increase the maximum Push ID by sending a MAX_PUSH_ID frame as the server fulfills or cancels server pushes.

The MAX_PUSH_ID frame has no defined flags.

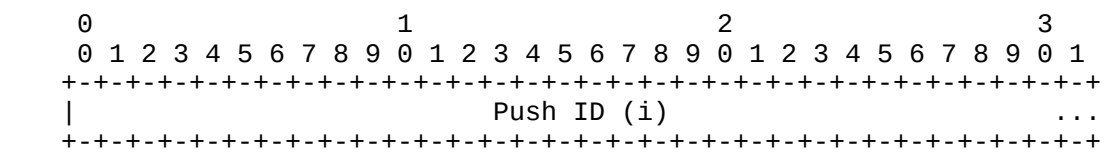


Figure 10: MAX_PUSH_ID frame payload

The MAX_PUSH_ID frame carries a single variable-length integer that identifies the maximum value for a Push ID that the server can use (see Section 4.2.6). A MAX_PUSH_ID frame cannot reduce the maximum Push ID; receipt of a MAX_PUSH_ID that contains a smaller value than previously received MUST be treated as a connection error of type HTTP_MALFORMED_FRAME.

A server MUST treat a MAX_PUSH_ID frame payload that does not contain a single variable-length integer as a connection error of type HTTP_MALFORMED_FRAME.

5. Connection Management

QUIC connections are persistent. All of the considerations in Section 9.1 of [RFC7540] apply to the management of QUIC connections.

HTTP clients are expected to use QUIC PING frames to keep connections open. Servers SHOULD NOT use PING frames to keep a connection open. A client SHOULD NOT use PING frames for this purpose unless there are responses outstanding for requests or server pushes. If the client is not expecting a response from the server, allowing an idle connection to time out (based on the idle_timeout transport parameter) is preferred over expending effort maintaining a connection that might not be needed. A gateway MAY use PING to maintain connections in anticipation of need rather than incur the latency cost of connection establishment to servers.

6. Error Handling

QUIC allows the application to abruptly terminate (reset) individual streams or the entire connection when an error is encountered. These are referred to as "stream errors" or "connection errors" and are described in more detail in [QUIC-TRANSPORT].

This section describes HTTP-specific error codes which can be used to express the cause of a connection or stream error.

6.1. HTTP/QUIC Error Codes

The following error codes are defined for use in QUIC RST_STREAM, STOP_SENDING, and CONNECTION_CLOSE frames when using HTTP/QUIC.

STOPPING (0x00): This value is reserved by the transport to be used in response to QUIC STOP_SENDING frames.

HTTP_NO_ERROR (0x01): No error. This is used when the connection or stream needs to be closed, but there is no error to signal.

HTTP_PUSH_REFUSED (0x02): The server has attempted to push content which the client will not accept on this connection.

HTTP_INTERNAL_ERROR (0x03): An internal error has occurred in the HTTP stack.

HTTP_PUSH_ALREADY_IN_CACHE (0x04): The server has attempted to push content which the client has cached.

HTTP_REQUEST_CANCELLED (0x05): The client no longer needs the requested data.

HTTP_HPACK_DECOMPRESSION_FAILED (0x06): HPACK failed to decompress a frame and cannot continue.

HTTP_CONNECT_ERROR (0x07): The connection established in response to a CONNECT request was reset or abnormally closed.

HTTP_EXCESSIVE_LOAD (0x08): The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

HTTP_VERSION_FALLBACK (0x09): The requested operation cannot be served over HTTP/QUIC. The peer should retry over HTTP/2.

HTTP_WRONG_STREAM (0x0A): A frame was received on stream where it is not permitted.

HTTP_PUSH_LIMIT_EXCEEDED (0x0B): A Push ID greater than the current maximum Push ID was referenced.

HTTP_DUPLICATE_PUSH (0x0C): A Push ID was referenced in two different stream headers.

HTTP_MALFORMED_FRAME (0x01XX): An error in a specific frame type. The frame type is included as the last octet of the error code. For example, an error in a MAX_PUSH_ID frame would be indicated with the code (0x10D).

7. Considerations for Transitioning from HTTP/2

HTTP/QUIC is strongly informed by HTTP/2, and bears many similarities. This section describes the approach taken to design HTTP/QUIC, points out important differences from HTTP/2, and describes how to map HTTP/2 extensions into HTTP/QUIC.

HTTP/QUIC begins from the premise that HTTP/2 code reuse is a useful feature, but not a hard requirement. HTTP/QUIC departs from HTTP/2 primarily where necessary to accommodate the differences in behavior between QUIC and TCP (lack of ordering, support for streams). We intend to avoid gratuitous changes which make it difficult or impossible to build extensions with the same semantics applicable to both protocols at once.

These departures are noted in this section.

7.1. Streams

HTTP/QUIC permits use of a larger number of streams (2^62-1) than HTTP/2. The considerations about exhaustion of stream identifier space apply, though the space is significantly larger such that it is likely that other limits in QUIC are reached first, such as the limit on the connection flow control window.

7.2. HTTP Frame Types

Many framing concepts from HTTP/2 can be elided away on QUIC, because the transport deals with them. Because frames are already on a stream, they can omit the stream number. Because frames do not block multiplexing (QUIC's multiplexing occurs below this layer), the support for variable-maximum-length packets can be removed. Because stream termination is handled by QUIC, an END_STREAM flag is not required.

Frame payloads are largely drawn from [RFC7540]. However, QUIC includes many features (e.g. flow control) which are also present in HTTP/2. In these cases, the HTTP mapping does not re-implement them. As a result, several HTTP/2 frame types are not required in HTTP/QUIC. Where an HTTP/2-defined frame is no longer used, the frame ID has been reserved in order to maximize portability between HTTP/2 and HTTP/QUIC implementations. However, even equivalent frames between the two mappings are not identical.

Many of the differences arise from the fact that HTTP/2 provides an absolute ordering between frames across all streams, while QUIC provides this guarantee on each stream only. As a result, if a frame

type makes assumptions that frames from different streams will still be received in the order sent, HTTP/QUIC will break them.

For example, implicit in the HTTP/2 prioritization scheme is the notion of in-order delivery of priority changes (i.e., dependency tree mutations): since operations on the dependency tree such as reparenting a subtree are not commutative, both sender and receiver must apply them in the same order to ensure that both sides have a consistent view of the stream dependency tree. HTTP/2 specifies priority assignments in PRIORITY frames and (optionally) in HEADERS frames. To achieve in-order delivery of priority changes in HTTP/QUIC, PRIORITY frames are sent on the control stream and the PRIORITY section is removed from the HEADERS frame.

Likewise, HPACK was designed with the assumption of in-order delivery. A sequence of encoded header blocks must arrive (and be decoded) at an endpoint in the same order in which they were encoded. This ensures that the dynamic state at the two endpoints remains in sync. As a result, HTTP/QUIC uses a modified version of HPACK, described in [QPACK].

Frame type definitions in HTTP/QUIC often use the QUIC variable-length integer encoding. In particular, Stream IDs use this encoding, which allow for a larger range of possible values than the encoding used in HTTP/2. Some frames in HTTP/QUIC use an identifier rather than a Stream ID (e.g. Push IDs in PRIORITY frames). Redefinition of the encoding of extension frame types might be necessary if the encoding includes a Stream ID.

Other than this issue, frame type HTTP/2 extensions are typically portable to QUIC simply by replacing Stream 0 in HTTP/2 with Stream 2 or 3 in HTTP/QUIC. HTTP/QUIC extensions will not assume ordering, but would not be harmed by ordering, and would be portable to HTTP/2 in the same manner.

Below is a listing of how each HTTP/2 frame type is mapped:

DATA (0x0): Padding is not defined in HTTP/QUIC frames. See Section 4.2.1.

HEADERS (0x1): As described above, the PRIORITY region of HEADERS is not supported. A separate PRIORITY frame MUST be used. Padding is not defined in HTTP/QUIC frames. See Section 4.2.2.

PRIORITY (0x2): As described above, the PRIORITY frame is sent on the control stream and can reference either a Stream ID or a Push ID. See Section 4.2.3.

RST_STREAM (0x3): RST_STREAM frames do not exist, since QUIC provides stream lifecycle management. The same code point is used for the CANCEL_PUSH frame (Section 4.2.4).

SETTINGS (0x4): SETTINGS frames are sent only at the beginning of the connection. See Section 4.2.5 and Section 7.3.

PUSH_PROMISE (0x5): The PUSH_PROMISE does not reference a stream; instead the push stream references the PUSH_PROMISE frame using a Push ID. See Section 4.2.6.

PING (0x6): PING frames do not exist, since QUIC provides equivalent functionality.

GOAWAY (0x7): GOAWAY is sent only from server to client and does not contain an error code. See Section 4.2.7.

WINDOW_UPDATE (0x8): WINDOW_UPDATE frames do not exist, since QUIC provides flow control.

CONTINUATION (0x9): CONTINUATION frames do not exist; instead, larger HEADERS/PUSH_PROMISE frames than HTTP/2 are permitted, and HEADERS frames can be used in series.

Frame types defined by extensions to HTTP/2 need to be separately registered for HTTP/QUIC if still applicable. The IDs of frames defined in [RFC7540] have been reserved for simplicity. See Section 9.3.

7.3. HTTP/2 SETTINGS Parameters

An important difference from HTTP/2 is that settings are sent once, at the beginning of the connection, and thereafter cannot change. This eliminates many corner cases around synchronization of changes.

Some transport-level options that HTTP/2 specifies via the SETTINGS frame are superseded by QUIC transport parameters in HTTP/QUIC. The HTTP-level options that are retained in HTTP/QUIC have the same value as in HTTP/2.

Below is a listing of how each HTTP/2 SETTINGS parameter is mapped:

SETTINGS_HEADER_TABLE_SIZE: See Section 4.2.5.2.

SETTINGS_ENABLE_PUSH: This is removed in favor of the MAX_PUSH_ID which provides a more granular control over server push.

SETTINGS_MAX_CONCURRENT_STREAMS: QUIC controls the largest open Stream ID as part of its flow control logic. Specifying SETTINGS_MAX_CONCURRENT_STREAMS in the SETTINGS frame is an error.

SETTINGS_INITIAL_WINDOW_SIZE: QUIC requires both stream and connection flow control window sizes to be specified in the initial transport handshake. Specifying SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS frame is an error.

SETTINGS_MAX_FRAME_SIZE: This setting has no equivalent in HTTP/QUIC. Specifying it in the SETTINGS frame is an error.

SETTINGS_MAX_HEADER_LIST_SIZE: See Section 4.2.5.2.

Settings need to be defined separately for HTTP/2 and HTTP/QUIC. The IDs of settings defined in [RFC7540] have been reserved for simplicity. See Section 9.4.

7.4. HTTP/2 Error Codes

QUIC has the same concepts of "stream" and "connection" errors that HTTP/2 provides. However, because the error code space is shared between multiple components, there is no direct portability of HTTP/2 error codes.

The HTTP/2 error codes defined in Section 7 of [RFC7540] map to the HTTP over QUIC error codes as follows:

NO_ERROR (0x0): HTTP_NO_ERROR in Section 6.1.

PROTOCOL_ERROR (0x1): No single mapping. See new HTTP_MALFORMED_FRAME error codes defined in Section 6.1.

INTERNAL_ERROR (0x2): HTTP_INTERNAL_ERROR in Section 6.1.

FLOW_CONTROL_ERROR (0x3): Not applicable, since QUIC handles flow control. Would provoke a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA from the QUIC layer.

SETTINGS_TIMEOUT (0x4): Not applicable, since no acknowledgement of SETTINGS is defined.

STREAM_CLOSED (0x5): Not applicable, since QUIC handles stream management. Would provoke a QUIC_STREAM_DATA_AFTER_TERMINATION from the QUIC layer.

FRAME_SIZE_ERROR (0x6) No single mapping. See new error codes defined in Section 6.1.

REFUSED_STREAM (0x7): Not applicable, since QUIC handles stream management. Would provoke a QUIC_TOO_MANY_OPEN_STREAMS from the QUIC layer.

CANCEL (0x8): HTTP_REQUEST_CANCELLED in Section 6.1.

COMPRESSION_ERROR (0x9): HTTP_HPACK_DECOMPRESSION_FAILED in Section 6.1.

CONNECT_ERROR (0xa): HTTP_CONNECT_ERROR in Section 6.1.

ENHANCE_YOUR_CALM (0xb): HTTP_EXCESSIVE_LOAD in Section 6.1.

INADEQUATE_SECURITY (0xc): Not applicable, since QUIC is assumed to provide sufficient security on all connections.

HTTP_1_1_REQUIRED (0xd): HTTP_VERSION_FALLBACK in Section 6.1.

Error codes need to be defined for HTTP/2 and HTTP/QUIC separately. See Section 9.5.

8. Security Considerations

The security considerations of HTTP over QUIC should be comparable to those of HTTP/2.

The modified SETTINGS format contains nested length elements, which could pose a security risk to an uncautious implementer. A SETTINGS frame parser MUST ensure that the length of the frame exactly matches the length of the settings it contains.

9. IANA Considerations

9.1. Registration of HTTP/QUIC Identification String

This document creates a new registration for the identification of HTTP/QUIC in the "Application Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [RFC7301].

The "hq" string identifies HTTP/QUIC:

Protocol: HTTP over QUIC

Identification Sequence: 0x68 0x71 ("hq")

Specification: This document

9.2. Registration of QUIC Version Hint Alt-Svc Parameter

This document creates a new registration for version-negotiation hints in the "Hypertext Transfer Protocol (HTTP) Alt-Svc Parameter" registry established in [RFC7838].

Parameter: "quic"

Specification: This document, Section 2.1.1

9.3. Frame Types

This document establishes a registry for HTTP/QUIC frame type codes. The "HTTP/QUIC Frame Type" registry manages an 8-bit space. The "HTTP/QUIC Frame Type" registry operates under either of the "IETF Review" or "IESG Approval" policies [RFC8126] for values between 0x00 and 0xef, with values between 0xf0 and 0xff being reserved for Experimental Use.

While this registry is separate from the "HTTP/2 Frame Type" registry defined in [RFC7540], it is preferable that the assignments parallel each other. If an entry is present in only one registry, every effort SHOULD be made to avoid assigning the corresponding value to an unrelated operation.

New entries in this registry require the following information:

Frame Type: A name or label for the frame type.

Code: The 8-bit code assigned to the frame type.

Specification: A reference to a specification that includes a description of the frame layout, its semantics, and flags that the frame type uses, including any parts of the frame that are conditionally present based on the value of flags.

The entries in the following table are registered by this document.

Frame Type	Code	Specification
DATA	0x0	Section 4.2.1
HEADERS	0x1	Section 4.2.2
PRIORITY	0x2	Section 4.2.3
CANCEL_PUSH	0x3	Section 4.2.4
SETTINGS	0x4	Section 4.2.5
PUSH_PROMISE	0x5	Section 4.2.6
Reserved	0x6	N/A
GOAWAY	0x7	Section 4.2.7
HEADER_ACK	0x8	Section 4.2.8
Reserved	0x9	N/A
MAX_PUSH_ID	0xD	Section 4.2.9

9.4. Settings Parameters

This document establishes a registry for HTTP/QUIC settings. The "HTTP/QUIC Settings" registry manages a 16-bit space. The "HTTP/QUIC Settings" registry operates under the "Expert Review" policy [RFC8126] for values in the range from 0x0000 to 0xefff, with values between and 0xf000 and 0xffff being reserved for Experimental Use. The designated experts are the same as those for the "HTTP/2 Settings" registry defined in [RFC7540].

While this registry is separate from the "HTTP/2 Settings" registry defined in [RFC7540], it is preferable that the assignments parallel each other. If an entry is present in only one registry, every effort SHOULD be made to avoid assigning the corresponding value to an unrelated operation.

New registrations are advised to provide the following information:

Name: A symbolic name for the setting. Specifying a setting name is optional.

Code: The 16-bit code assigned to the setting.

Specification: An optional reference to a specification that describes the use of the setting.

The entries in the following table are registered by this document.

Setting Name	Code	Specification
HEADER_TABLE_SIZE	0x1	Section 4.2.5.2
Reserved	0x2	N/A
Reserved	0x3	N/A
Reserved	0x4	N/A
Reserved	0x5	N/A
MAX_HEADER_LIST_SIZE	0x6	Section 4.2.5.2

9.5. Error Codes

This document establishes a registry for HTTP/QUIC error codes. The "HTTP/QUIC Error Code" registry manages a 16-bit space. The "HTTP/QUIC Error Code" registry operates under the "Expert Review" policy [RFC8126].

Registrations for error codes are required to include a description of the error code. An expert reviewer is advised to examine new registrations for possible duplication with existing error codes. Use of existing registrations is to be encouraged, but not mandated.

New registrations are advised to provide the following information:

Name: A name for the error code. Specifying an error code name is optional.

Code: The 16-bit error code value.

Description: A brief description of the error code semantics, longer if no detailed specification is provided.

Specification: An optional reference for a specification that defines the error code.

The entries in the following table are registered by this document.

Name	Code	Description	Specification
STOPPING	0x0000	Reserved by QUIC	[QUIC-TRANSPORT]
HTTP_NO_ERROR	0x0001	No error	Section 6.1
HTTP_PUSH_REFUSED	0x0002	Client refused pushed content	Section 6.1
HTTP_INTERNAL_ERROR	0x0003	Internal error	Section 6.1
HTTP_PUSH_ALREADY_IN_CACHE	0x0004	Pushed content already cached	Section 6.1
HTTP_REQUEST_CANCELLED	0x0005	Data no longer needed	Section 6.1
HTTP_HPACK_DECOMPRESSION_FAILED	0x0006	HPACK cannot continue	Section 6.1
HTTP_CONNECT_ERROR	0x0007	TCP reset or error on CONNECT request	Section 6.1
HTTP_EXCESSIVE_LOAD	0x0008	Peer generating excessive load	Section 6.1
HTTP_VERSION_FALLBACK	0x0009	Retry over HTTP/2	Section 6.1
HTTP_WRONG_STREAM	0x000A	A frame was sent on the wrong stream	Section 6.1

HTTP_PUSH_LIMIT_EXCEEDED	0x000B	Maximum Push ID exceeded	Section 6.1
HTTP_DUPLICATE_PUSH	0x000C	Push ID was fulfilled multiple times	Section 6.1
HTTP_MALFORMED_FRAME	0x01XX	Error in frame formatting or use	Section 6.1

10. References

10.1. Normative References

[QPACK] Krasic, C., Bishop, M., and A. Frindell, Ed., "QPACK: Header Compression for HTTP over QUIC", draft-ietf-quic-qpack-00 (work in progress), April 2018.

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-11 (work in progress), April 2018.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <https://www.rfc-editor.org/info/rfc7838>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

10.2. Informative References

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

10.3. URIs

[1] https://mailarchive.ietf.org/arch/search/?email_list=quic

[2] https://github.com/quicwg

[3] https://github.com/quicwg/base-drafts/labels/-http

Appendix A. Contributors

The original authors of this specification were Robbie Shade and Mike Warres.

A substantial portion of Mike's contribution was supported by Microsoft during his employment there.

Appendix B. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

B.1. Since draft-ietf-quic-http-10

- Settings need to be remembered when attempting and accepting 0-RTT (#1157, #1207)

B.2. Since draft-ietf-quic-http-09

- Selected QCRAM for header compression (#228, #1117)
- The server_name TLS extension is now mandatory (#296, #495)
- Specified handling of unsupported versions in Alt-Svc (#1093, #1097)

B.3. Since draft-ietf-quic-http-08

- Clarified connection coalescing rules (#940, #1024)

B.4. Since draft-ietf-quic-http-07

- Changes for integer encodings in QUIC (#595,#905)
- Use unidirectional streams as appropriate (#515, #240, #281, #886)
- Improvement to the description of GOAWAY (#604, #898)
- Improve description of server push usage (#947, #950, #957)

B.5. Since draft-ietf-quic-http-06

- Track changes in QUIC error code usage (#485)

B.6. Since draft-ietf-quic-http-05

- Made push ID sequential, add MAX_PUSH_ID, remove SETTINGS_ENABLE_PUSH (#709)
- Guidance about keep-alive and QUIC PINGs (#729)
- Expanded text on GOAWAY and cancellation (#757)

B.7. Since draft-ietf-quic-http-04

- Cite RFC 5234 (#404)
- Return to a single stream per request (#245,#557)
- Use separate frame type and settings registries from HTTP/2 (#81)
- SETTINGS_ENABLE_PUSH instead of SETTINGS_DISABLE_PUSH (#477)
- Restored GOAWAY (#696)
- Identify server push using Push ID rather than a stream ID (#702,#281)
- DATA frames cannot be empty (#700)

B.8. Since draft-ietf-quic-http-03

None.

B.9. Since draft-ietf-quic-http-02

- Track changes in transport draft

B.10. Since draft-ietf-quic-http-01

- SETTINGS changes (#181):
  - SETTINGS can be sent only once at the start of a connection; no changes thereafter
  - SETTINGS_ACK removed
  - Settings can only occur in the SETTINGS frame a single time
  - Boolean format updated
- Alt-Svc parameter changed from "v" to "quic"; format updated (#229)
- Closing the connection control stream or any message control stream is a fatal error (#176)
- HPACK Sequence counter can wrap (#173)
- 0-RTT guidance added

- Guide to differences from HTTP/2 and porting HTTP/2 extensions added (#127,#242)

B.11. Since draft-ietf-quic-http-00

- Changed "HTTP/2-over-QUIC" to "HTTP/QUIC" throughout (#11,#29)
- Changed from using HTTP/2 framing within Stream 3 to new framing format and two-stream-per-request model (#71,#72,#73)
- Adopted SETTINGS format from draft-bishop-httpbis-extended-settings-01
- Reworked SETTINGS_ACK to account for indeterminate inter-stream order (#75)
- Described CONNECT pseudo-method (#95)
- Updated ALPN token and Alt-Svc guidance (#13,#87)
- Application-layer-defined error codes (#19,#74)

B.12. Since draft-shade-quic-http2-mapping-00

- Adopted as base for draft-ietf-quic-http
- Updated authors/editors list

Author’s Address

Mike Bishop (editor)
Akamai

Email: mbishop@evequefou.be

QUIC
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

J. Iyengar, Ed.
I. Swett, Ed.
Google

March 13, 2017

QUIC Loss Detection and Congestion Control
draft-ietf-quic-recovery-02

Abstract

QUIC is a new multiplexed and secure transport atop UDP. QUIC builds on decades of transport and security experience, and implements mechanisms that make it attractive as a modern general-purpose transport. QUIC implements the spirit of known TCP loss detection mechanisms, described in RFCs, various Internet-drafts, and also those prevalent in the Linux TCP implementation. This document describes QUIC loss detection and congestion control, and attributes the TCP equivalent in RFCs, Internet-drafts, academic papers, and TCP implementations.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic .

Working Group information can be found at https://github.com/quicwg ; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/recovery .

Status of This Memo

Table of Contents

1. Introduction

QUIC is a new multiplexed and secure transport atop UDP. QUIC builds on decades of transport and security experience, and implements mechanisms that make it attractive as a modern general-purpose transport. The QUIC protocol is described in [QUIC-TRANSPORT].

QUIC implements the spirit of known TCP loss recovery mechanisms, described in RFCs, various Internet-drafts, and also those prevalent in the Linux TCP implementation. This document describes QUIC congestion control and loss recovery, and where applicable, attributes the TCP equivalent in RFCs, Internet-drafts, academic papers, and/or TCP implementations.

This document first describes pre-requisite parts of the QUIC transmission machinery, then discusses QUIC's default congestion control and loss detection mechanisms, and finally lists the various TCP mechanisms that QUIC loss detection implements (in spirit.)

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

2. Design of the QUIC Transmission Machinery

All transmissions in QUIC are sent with a packet-level header, which includes a packet sequence number (referred to below as a packet number). These packet numbers never repeat in the lifetime of a connection, and are monotonically increasing, which makes duplicate detection trivial. This fundamental design decision obviates the need for disambiguating between transmissions and retransmissions and eliminates significant complexity from QUIC's interpretation of TCP loss detection mechanisms.

Every packet may contain several frames. We outline the frames that are important to the loss detection and congestion control machinery below.

- Retransmittable frames are frames requiring reliable delivery. The most common are STREAM frames, which typically contain application data.

- Crypto handshake data is also sent as STREAM data, and uses the reliability machinery of QUIC underneath.

- ACK frames contain acknowledgment information. QUIC uses a SACK-based scheme, where acks express up to 256 ranges. The ACK frame also includes a receive timestamp for each packet newly acked.

2.1. Relevant Differences Between QUIC and TCP

There are some notable differences between QUIC and TCP which are important for reasoning about the differences between the loss recovery mechanisms employed by the two protocols. We briefly describe these differences below.

2.1.1. Monotonically Increasing Packet Numbers

TCP conflates transmission sequence number at the sender with delivery sequence number at the receiver, which results in retransmissions of the same data carrying the same sequence number, and consequently to problems caused by "retransmission ambiguity". QUIC separates the two: QUIC uses a packet sequence number (referred to as the "packet number") for transmissions, and any data that is to be delivered to the receiving application(s) is sent in one or more streams, with stream offsets encoded within STREAM frames inside of packets that determine delivery order.

QUIC’s packet number is strictly increasing, and directly encodes transmission order. A higher QUIC packet number signifies that the packet was sent later, and a lower QUIC packet number signifies that the packet was sent earlier. When a packet containing frames is deemed lost, QUIC rebundles necessary frames in a new packet with a new packet number, removing ambiguity about which packet is acknowledged when an ACK is received. Consequently, more accurate RTT measurements can be made, spurious retransmissions are trivially detected, and mechanisms such as Fast Retransmit can be applied universally, based only on packet number.

This design point significantly simplifies loss detection mechanisms for QUIC. Most TCP mechanisms implicitly attempt to infer transmission ordering based on TCP sequence numbers - a non-trivial task, especially when TCP timestamps are not available.

2.1.2. No Reneging

QUIC ACKs contain information that is equivalent to TCP SACK, but QUIC does not allow any acked packet to be reneged, greatly simplifying implementations on both sides and reducing memory pressure on the sender.

2.1.3. More ACK Ranges

QUIC supports up to 256 ACK ranges, opposed to TCP's 3 SACK ranges. In high loss environments, this speeds recovery.

2.1.4. Explicit Correction For Delayed Acks

QUIC ACKs explicitly encode the delay incurred at the receiver between when a packet is received and when the corresponding ACK is sent. This allows the receiver of the ACK to adjust for receiver delays, specifically the delayed ack timer, when estimating the path RTT. This mechanism also allows a receiver to measure and report the delay from when a packet was received by the OS kernel, which is useful in receivers which may incur delays such as context-switch latency before a userspace QUIC receiver processes a received packet.

3. Loss Detection

We now describe QUIC's loss detection as functions that should be called on packet transmission, when a packet is acked, and timer expiration events.

3.1. Constants of interest

Constants used in loss recovery and congestion control are based on a combination of RFCs, papers, and common practice. Some may need to be changed or negotiated in order to better suit a variety of environments.

kMaxTLPs (default 2): Maximum number of tail loss probes before an RTO fires.

kReorderingThreshold (default 3): Maximum reordering in packet number space before FACK style loss detection considers a packet lost.

kTimeReorderingFraction (default 1/8): Maximum reordering in time sapce before time based loss detection considers a packet lost. In fraction of an RTT.

kMinTLPTimeout (default 10ms): Minimum time in the future a tail loss probe alarm may be set for.

kMinRTOTimeout (default 200ms): Minimum time in the future an RTO alarm may be set for.

kDelayedAckTimeout (default 25ms): The length of the peer's delayed ack timer.

kDefaultInitialRtt (default 100ms): The default RTT used before an RTT sample is taken.

3.2. Variables of interest

We first describe the variables required to implement the loss detection mechanisms described in this section.

loss_detection_alarm: Multi-modal alarm used for loss detection.

handshake_count: The number of times the handshake packets have been retransmitted without receiving an ack.

tlp_count: The number of times a tail loss probe has been sent without receiving an ack.

rto_count: The number of times an rto has been sent without receiving an ack.

smoothed_rtt: The smoothed RTT of the connection, computed as described in [RFC6298]

rttvar: The RTT variance, computed as described in [RFC6298]

initial_rtt: The initial RTT used before any RTT measurements have been made.

reordering_threshold: The largest delta between the largest acked retransmittable packet and a packet containing retransmittable frames before it’s declared lost.

time_reordering_fraction: The reordering window as a fraction of max(smoothed_rtt, latest_rtt).

loss_time: The time at which the next packet will be considered lost based on early transmit or exceeding the reordering window in time.

sent_packets: An association of packet numbers to information about them, including a number field indicating the packet number, a time field indicating the time a packet was sent, and a bytes field indicating the packet’s size. sent_packets is ordered by packet number, and packets remain in sent_packets until acknowledged or lost.

3.3. Initialization

At the beginning of the connection, initialize the loss detection variables as follows:

```
loss_detection_alarm.reset()
handshake_count = 0
tlp_count = 0
rto_count = 0
if (UsingTimeLossDetection())
  reordering_threshold = infinite
  time_reordering_fraction = kTimeReorderingFraction
else:
  reordering_threshold = kReorderingThreshold
  time_reordering_fraction = infinite
loss_time = 0
smoothed_rtt = 0
rttvar = 0
initial_rtt = kDefaultInitialRtt
```

3.4. On Sending a Packet

After any packet is sent, be it a new transmission or a rebundled transmission, the following OnPacketSent function is called. The parameters to OnPacketSent are as follows:

- packet_number: The packet number of the sent packet.

- is_retransmittble: A boolean that indicates whether the packet contains at least one frame requiring reliable deliver. The retransmittability of various QUIC frames is described in [QUIC-TRANSPORT]. If false, it is still acceptable for an ack to be received for this packet. However, a caller MUST NOT set is_retransmittable to true if an ack is not expected.

- sent_bytes: The number of bytes sent in the packet.

Pseudocode for OnPacketSent follows:

```
OnPacketSent(packet_number, is_retransmittable, sent_bytes):
  sent_packets[packet_number].packet_number = packet_number
  sent_packets[packet_number].time = now
  if is_retransmittable:
    sent_packets[packet_number].bytes = sent_bytes
    SetLossDetectionAlarm()
```

3.5. On Ack Receipt

When an ack is received, it may acknowledge 0 or more packets.

Pseudocode for OnAckReceived and UpdateRtt follow:

```
OnAckReceived(ack):
  // If the largest acked is newly acked, update the RTT.
  if (sent_packets[ack.largest_acked]):
    rtt_sample = now - sent_packets[ack.largest_acked].time
    if (rtt_sample > ack.ack_delay):
      rtt_sample -= ack.delay
    UpdateRtt(rtt_sample)
  // Find all newly acked packets.
  for acked_packet_number in DetermineNewlyAckedPackets():
    OnPacketAcked(acked_packet_number)

  DetectLostPackets(ack.largest_acked_packet)
  SetLossDetectionAlarm()


UpdateRtt(rtt_sample):
  // Based on {{RFC6298}}.
  if (smoothed_rtt == 0):
    smoothed_rtt = rtt_sample
    rttvar = rtt_sample / 2
  else:
    rttvar = 3/4 * rttvar + 1/4 * (smoothed_rtt - rtt_sample)
    smoothed_rtt = 7/8 * smoothed_rtt + 1/8 * rtt_sample
```

3.6. On Packet Acknowledgment

When a packet is acked for the first time, the following OnPacketAcked function is called. Note that a single ACK frame may newly acknowledge several packets. OnPacketAcked must be called once for each of these newly acked packets.

OnPacketAcked takes one parameter, acked_packet, which is the packet number of the newly acked packet, and returns a list of packet numbers that are detected as lost.

Pseudocode for OnPacketAcked follows:

```
OnPacketAcked(acked_packet_number):
  handshake_count = 0
  tlp_count = 0
  rto_count = 0
  sent_packets.remove(acked_packet_number)
```

3.7. Setting the Loss Detection Alarm

QUIC loss detection uses a single alarm for all timer-based loss detection. The duration of the alarm is based on the alarm's mode, which is set in the packet and timer events further below. The function SetLossDetectionAlarm defined below shows how the single timer is set based on the alarm mode.

3.7.1. Handshake Packets

The initial flight has no prior RTT sample. A client SHOULD remember the previous RTT it observed when resumption is attempted and use that for an initial RTT value. If no previous RTT is available, the initial RTT defaults to 200ms. Once an RTT measurement is taken, it MUST replace initial_rtt.

Endpoints MUST retransmit handshake frames if not acknowledged within a time limit. This time limit will start as the largest of twice the rtt value and MinTLPTimeout. Each consecutive handshake retransmission doubles the time limit, until an acknowledgement is received.

Handshake frames may be cancelled by handshake state transitions. In particular, all non-protected frames SHOULD be no longer be transmitted once packet protection is available.

When stateless rejects are in use, the connection is considered immediately closed once a reject is sent, so no timer is set to retransmit the reject.

Version negotiation packets are always stateless, and MUST be sent once per per handshake packet that uses an unsupported QUIC version, and MAY be sent in response to 0RTT packets.

3.7.2. Tail Loss Probe and Retransmission Timeout

Tail loss probes [I-D.dukkipati-tcpm-tcp-loss-probe] and retransmission timeouts[RFC6298] are an alarm based mechanism to recover from cases when there are outstanding retransmittable packets, but an acknowledgement has not been received in a timely manner.

3.7.3. Early Retransmit

Early retransmit [RFC5827] is implemented with a 1/4 RTT timer. It is part of QUIC's time based loss detection, but is always enabled, even when only packet reordering loss detection is enabled.

3.7.4. Pseudocode

Pseudocode for SetLossDetectionAlarm follows:

```
 SetLossDetectionAlarm():
    if (retransmittable packets are not outstanding):
      loss_detection_alarm.cancel();
      return

    if (handshake packets are outstanding):
      // Handshake retransmission alarm.
      if (smoothed_rtt == 0):
        alarm_duration = 2 * initial_rtt
      else:
        alarm_duration = 2 * smoothed_rtt
      alarm_duration = max(alarm_duration, kMinTLPTimeout)
      alarm_duration = alarm_duration << handshake_count
    else if (loss_time != 0):
      // Early retransmit timer or time loss detection.
      alarm_duration = loss_time - now
    else if (tlp_count < kMaxTLPs):
      // Tail Loss Probe
      if (retransmittable_packets_outstanding = 1):
        alarm_duration = 1.5 * smoothed_rtt + kDelayedAckTimeout
      else:
        alarm_duration = kMinTLPTimeout
      alarm_duration = max(alarm_duration, 2 * smoothed_rtt)
    else:
      // RTO alarm
      if (rto_count = 0):
        alarm_duration = smoothed_rtt + 4 * rttvar
        alarm_duration = max(alarm_duration, kMinRTOTimeout)
      else:
        alarm_duration = loss_detection_alarm.get_delay() << 1

    loss_detection_alarm.set(now + alarm_duration)
```

3.8. On Alarm Firing

QUIC uses one loss recovery alarm, which when set, can be in one of several modes. When the alarm fires, the mode determines the action to be performed.

Pseudocode for OnLossDetectionAlarm follows:

```
OnLossDetectionAlarm():
  if (handshake packets are outstanding):
    // Handshake retransmission alarm.
    RetransmitAllHandshakePackets();
    handshake_count++;
  // TODO: Clarify early retransmit and time loss.
  else if (loss_time != 0):
    // Early retransmit or Time Loss Detection
    DetectLostPackets(largest_acked_packet)
  else if (tlp_count < kMaxTLPs):
    // Tail Loss Probe.
    if (HasNewDataToSend()):
      SendOnePacketOfNewData()
    else:
      RetransmitOldestPacket()
    tlp_count++
  else:
    // RTO.
    RetransmitOldestTwoPackets()
    rto_count++

  SetLossDetectionAlarm()
```

3.9. Detecting Lost Packets

Packets in QUIC are only considered lost once a larger packet number is acknowledged. DetectLostPackets is called every time an ack is received. If the loss detection alarm fires and the loss_time is set, the previous largest acked packet is supplied.

3.9.1. Handshake Packets

The receiver MUST ignore unprotected packets that ack protected packets. The receiver MUST trust protected acks for unprotected packets, however. Aside from this, loss detection for handshake packets when an ack is processed is identical to other packets.

3.9.2. Pseudocode

DetectLostPackets takes one parameter, acked, which is the largest acked packet.

Pseudocode for DetectLostPackets follows:

```
DetectLostPackets(largest_acked):
  loss_time = 0
  lost_packets = {}
  delay_until_lost = infinite;
  if (time_reordering_fraction != infinite):
    delay_until_lost =
      (1 + time_reordering_fraction) * max(latest_rtt, smoothed_rtt)
  else if (largest_acked.packet_number == largest_sent_packet):
    // Early retransmit alarm.
    delay_until_lost = 9/8 * max(latest_rtt, smoothed_rtt)
  foreach (unacked less than largest_acked.packet_number):
    time_since_sent = now() - unacked.time_sent
    packet_delta = largest_acked.packet_number - unacked.packet_number
    if (time_since_sent > delay_until_lost):
      lost_packets.insert(unacked)
    else if (packet_delta > reordering_threshold)
      lost_packets.insert(unacked)
    else if (loss_time == 0 && delay_until_lost != infinite):
      loss_time = delay_until_lost - time_since_sent

  // Inform the congestion controller of lost packets and
  // lets it decide whether to retransmit immediately.
  OnPacketsLost(lost_packets)
  foreach (packet in lost_packets)
    sent_packets.remove(packet.packet_number)
```

4. Congestion Control

(describe NewReno-style congestion control [RFC6582] for QUIC.) (describe appropriate byte counting.) (define recovery based on packet numbers.) (describe min_rtt based hystart.) (describe how QUIC's F-RTO [RFC5682] delays reducing CWND.) (describe PRR [RFC6937])

5. IANA Considerations

This document has no IANA actions. Yet.

6. References

6.1. Normative References

[QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport".

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

6.2. Informative References

[I-D.dukkipati-tcpm-tcp-loss-probe] Dukkipati, N., Cardwell, N., Cheng, Y., and M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses", draft-dukkipati-tcpm-tcp-loss-probe-01 (work in progress), February 2013.

[RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, DOI 10.17487/RFC5682, September 2009, <http://www.rfc-editor.org/info/rfc5682>.

[RFC5827] Allman, M., Avrachenkov, K., Ayesta, U., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, DOI 10.17487/RFC5827, May 2010, <http://www.rfc-editor.org/info/rfc5827>.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <http://www.rfc-editor.org/info/rfc6298>.

[RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, DOI 10.17487/RFC6582, April 2012, <http://www.rfc-editor.org/info/rfc6582>.

[RFC6937] Mathis, M., Dukkipati, N., and Y. Cheng, "Proportional Rate Reduction for TCP", RFC 6937, DOI 10.17487/RFC6937, May 2013, <http://www.rfc-editor.org/info/rfc6937>.

Appendix A. Acknowledgments

Appendix B. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

B.1. Since draft-ietf-quic-recovery-01

- Changes initial default RTT to 100ms
- Added time-based loss detection and fixes early retransmit
- Clarified loss recovery for handshake packets
- Fixed references and made TCP references informative

B.2. Since draft-ietf-quic-recovery-00:

- Improved description of constants and ACK behavior

B.3. Since draft-iyengar-quic-loss-recovery-01:

- Adopted as base for draft-ietf-quic-recovery.
- Updated authors/editors list.
- Added table of contents.

Authors' Addresses

Jana Iyengar (editor)
Google

Email: jri@google.com

Ian Swett (editor)
Google

Email: ianswett@google.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: October 19, 2018

J. Iyengar, Ed.
Fastly
I. Swett, Ed.
Google

April 17, 2018

QUIC Loss Detection and Congestion Control
draft-ietf-quic-recovery-11

Abstract

This document describes loss detection and congestion control mechanisms for QUIC.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic [1].

Working Group information can be found at https://github.com/quicwg [2]; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/-recovery [3].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 19, 2018.

Copyright Notice

Table of Contents

1. Introduction

QUIC is a new multiplexed and secure transport atop UDP. QUIC builds on decades of transport and security experience, and implements mechanisms that make it attractive as a modern general-purpose transport. The QUIC protocol is described in [QUIC-TRANSPORT].

QUIC implements the spirit of known TCP loss recovery mechanisms, described in RFCs, various Internet-drafts, and also those prevalent in the Linux TCP implementation. This document describes QUIC congestion control and loss recovery, and where applicable, attributes the TCP equivalent in RFCs, Internet-drafts, academic papers, and/or TCP implementations.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Design of the QUIC Transmission Machinery

All transmissions in QUIC are sent with a packet-level header, which includes a packet sequence number (referred to below as a packet number). These packet numbers never repeat in the lifetime of a connection, and are monotonically increasing, which prevents ambiguity. This fundamental design decision obviates the need for disambiguating between transmissions and retransmissions and eliminates significant complexity from QUIC's interpretation of TCP loss detection mechanisms.

Every packet may contain several frames. We outline the frames that are important to the loss detection and congestion control machinery below.

- Retransmittable frames are those that count towards bytes in flight and need acknowledgement. The most common are STREAM frames, which typically contain application data.

- Retransmittable packets are those that contain at least one retransmittable frame.

- Crypto handshake data is sent on stream 0, and uses the reliability machinery of QUIC underneath.

- ACK frames contain acknowledgment information. ACK frames contain one or more ranges of acknowledged packets.

2.1. Relevant Differences Between QUIC and TCP

Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones. Protocol differences between QUIC and TCP however contribute to algorithmic differences. We briefly describe these protocol differences below.

2.1.1. Monotonically Increasing Packet Numbers

TCP conflates transmission sequence number at the sender with delivery sequence number at the receiver, which results in retransmissions of the same data carrying the same sequence number, and consequently to problems caused by "retransmission ambiguity". QUIC separates the two: QUIC uses a packet number for transmissions, and any data that is to be delivered to the receiving application(s) is sent in one or more streams, with delivery order determined by stream offsets encoded within STREAM frames.

QUIC's packet number is strictly increasing, and directly encodes transmission order. A higher QUIC packet number signifies that the packet was sent later, and a lower QUIC packet number signifies that the packet was sent earlier. When a packet containing frames is deemed lost, QUIC rebundles necessary frames in a new packet with a new packet number, removing ambiguity about which packet is acknowledged when an ACK is received. Consequently, more accurate RTT measurements can be made, spurious retransmissions are trivially detected, and mechanisms such as Fast Retransmit can be applied universally, based only on packet number.

This design point significantly simplifies loss detection mechanisms for QUIC. Most TCP mechanisms implicitly attempt to infer transmission ordering based on TCP sequence numbers - a non-trivial task, especially when TCP timestamps are not available.

2.1.2. No Reneging

QUIC ACKs contain information that is similar to TCP SACK, but QUIC does not allow any acked packet to be reneged, greatly simplifying implementations on both sides and reducing memory pressure on the sender.

2.1.3. More ACK Ranges

QUIC supports many ACK ranges, opposed to TCP's 3 SACK ranges. In high loss environments, this speeds recovery, reduces spurious retransmits, and ensures forward progress without relying on timeouts.

2.1.4. Explicit Correction For Delayed ACKs

QUIC ACKs explicitly encode the delay incurred at the receiver between when a packet is received and when the corresponding ACK is sent. This allows the receiver of the ACK to adjust for receiver delays, specifically the delayed ack timer, when estimating the path RTT. This mechanism also allows a receiver to measure and report the

delay from when a packet was received by the OS kernel, which is useful in receivers which may incur delays such as context-switch latency before a userspace QUIC receiver processes a received packet.

3. Loss Detection

QUIC senders use both ack information and timeouts to detect lost packets, and this section provides a description of these algorithms. Estimating the network round-trip time (RTT) is critical to these algorithms and is described first.

3.1. Computing the RTT estimate

RTT is calculated when an ACK frame arrives by computing the difference between the current time and the time the largest newly acked packet was sent. If no packets are newly acknowledged, RTT cannot be calculated. When RTT is calculated, the ack delay field from the ACK frame SHOULD be subtracted from the RTT as long as the result is larger than the Min RTT. If the result is smaller than the min_rtt, the RTT should be used, but the ack delay field should be ignored.

Like TCP, QUIC calculates both smoothed RTT and RTT variance similar to those specified in [RFC6298].

Min RTT is the minimum RTT measured over the connection, prior to adjusting by ack delay. Ignoring ack delay for min RTT prevents intentional or unintentional underestimation of min RTT, which in turn prevents underestimating smoothed RTT.

3.2. Ack-based Detection

Ack-based loss detection implements the spirit of TCP’s Fast Retransmit [RFC5681], Early Retransmit [RFC5827], FACK, and SACK loss recovery [RFC6675]. This section provides an overview of how these algorithms are implemented in QUIC.

3.2.1. Fast Retransmit

An unacknowledged packet is marked as lost when an acknowledgment is received for a packet that was sent a threshold number of packets (kReorderingThreshold) after the unacknowledged packet. Receipt of the ack indicates that a later packet was received, while kReorderingThreshold provides some tolerance for reordering of packets in the network.

The RECOMMENDED initial value for kReorderingThreshold is 3.

We derive this default from recommendations for TCP loss recovery [RFC5681] [RFC6675]. It is possible for networks to exhibit higher degrees of reordering, causing a sender to detect spurious losses. Detecting spurious losses leads to unnecessary retransmissions and may result in degraded performance due to the actions of the congestion controller upon detecting loss. Implementers MAY use algorithms developed for TCP, such as TCP-NCR [RFC4653], to improve QUIC's reordering resilience, though care should be taken to map TCP specifics to QUIC correctly. Similarly, using time-based loss detection to deal with reordering, such as in PR-TCP, should be more readily usable in QUIC. Making QUIC deal with such networks is important open research, and implementers are encouraged to explore this space.

3.2.2. Early Retransmit

Unacknowledged packets close to the tail may have fewer than kReorderingThreshold retransmittable packets sent after them. Loss of such packets cannot be detected via Fast Retransmit. To enable ack-based loss detection of such packets, receipt of an acknowledgment for the last outstanding retransmittable packet triggers the Early Retransmit process, as follows.

If there are unacknowledged retransmittable packets still pending, they should be marked as lost. To compensate for the reduced reordering resilience, the sender SHOULD set an alarm for a small period of time. If the unacknowledged retransmittable packets are not acknowledged during this time, then these packets MUST be marked as lost.

An endpoint SHOULD set the alarm such that a packet is marked as lost no earlier than 1.25 * max(SRTT, latest_RTT) since when it was sent.

Using max(SRTT, latest_RTT) protects from the two following cases:

- the latest RTT sample is lower than the SRTT, perhaps due to reordering where packet whose ack triggered the Early Retransit process encountered a shorter path;

- the latest RTT sample is higher than the SRTT, perhaps due to a sustained increase in the actual RTT, but the smoothed SRTT has not yet caught up.

The 1.25 multiplier increases reordering resilience. Implementers MAY experiment with using other multipliers, bearing in mind that a lower multiplier reduces reordering resilience and increases spurious retransmissions, and a higher multipler increases loss recovery delay.

This mechanism is based on Early Retransmit for TCP [RFC5827]. However, [RFC5827] does not include the alarm described above. Early Retransmit is prone to spurious retransmissions due to its reduced reordering resilence without the alarm. This observation led Linux TCP implementers to implement an alarm for TCP as well, and this document incorporates this advancement.

3.3. Timer-based Detection

Timer-based loss detection implements a handshake retransmission timer that is optimized for QUIC as well as the spirit of TCP's Tail Loss Probe and Retransmission Timeout mechanisms.

3.3.1. Handshake Timeout

Handshake packets, which contain STREAM frames for stream 0, are critical to QUIC transport and crypto negotiation, so a separate alarm is used for them.

The initial handshake timeout SHOULD be set to twice the initial RTT.

At the beginning, there are no prior RTT samples within a connection. Resumed connections over the same network SHOULD use the previous connection's final smoothed RTT value as the resumed connection's initial RTT.

If no previous RTT is available, or if the network changes, the initial RTT SHOULD be set to 100ms.

When a handshake packet is sent, the sender SHOULD set an alarm for the handshake timeout period.

When the alarm fires, the sender MUST retransmit all unacknowledged handshake data, by calling RetransmitAllUnackedHandshakeData(). On each consecutive firing of the handshake alarm, the sender SHOULD double the handshake timeout and set an alarm for this period.

When an acknowledgement is received for a handshake packet, the new RTT is computed and the alarm SHOULD be set for twice the newly computed smoothed RTT.

Handshake data may be cancelled by handshake state transitions. In particular, all non-protected data SHOULD no longer be transmitted once packet protection is available.

(TODO: Work this section some more. Add text on client vs. server, and on stateless retry.)

3.3.2. Tail Loss Probe

The algorithm described in this section is an adaptation of the Tail Loss Probe algorithm proposed for TCP [TLP].

A packet sent at the tail is particularly vulnerable to slow loss detection, since acks of subsequent packets are needed to trigger ack-based detection. To ameliorate this weakness of tail packets, the sender schedules an alarm when the last retransmittable packet before quiescence is transmitted. When this alarm fires, a Tail Loss Probe (TLP) packet is sent to evoke an acknowledgement from the receiver.

The alarm duration, or Probe Timeout (PTO), is set based on the following conditions:

- PTO SHOULD be scheduled for max(1.5*SRTT+MaxAckDelay, kMinTLPTimeout)

- If RTO (Section 3.3.3) is earlier, schedule a TLP alarm in its place. That is, PTO SHOULD be scheduled for min(RTO, PTO).

MaxAckDelay is the maximum ack delay supplied in an incoming ACK frame. MaxAckDelay excludes ack delays that aren't included in an RTT sample because they're too large and excludes those which reference an ack-only packet.

QUIC diverges from TCP by calculating MaxAckDelay dynamically, instead of assuming a constant delayed ack timeout for all connections. QUIC includes this in all probe timeouts, because it assume the ack delay may come into play, regardless of the number of packets outstanding. TCP's TLP assumes if at least 2 packets are outstanding, acks will not be delayed.

A PTO value of at least 1.5*SRTT ensures that the ACK is overdue. The 1.5 is based on [TLP], but implementations MAY experiment with other constants.

To reduce latency, it is RECOMMENDED that the sender set and allow the TLP alarm to fire twice before setting an RTO alarm. In other words, when the TLP alarm fires the first time, a TLP packet is sent, and it is RECOMMENDED that the TLP alarm be scheduled for a second time. When the TLP alarm fires the second time, a second TLP packet is sent, and an RTO alarm SHOULD be scheduled Section 3.3.3.

A TLP packet SHOULD carry new data when possible. If new data is unavailable or new data cannot be sent due to flow control, a TLP packet MAY retransmit unacknowledged data to potentially reduce

recovery time. Since a TLP alarm is used to send a probe into the network prior to establishing any packet loss, prior unacknowledged packets SHOULD NOT be marked as lost when a TLP alarm fires.

A sender may not know that a packet being sent is a tail packet. Consequently, a sender may have to arm or adjust the TLP alarm on every sent retransmittable packet.

3.3.3. Retransmission Timeout

A Retransmission Timeout (RTO) alarm is the final backstop for loss detection. The algorithm used in QUIC is based on the RTO algorithm for TCP [RFC5681] and is additionally resilient to spurious RTO events [RFC5682].

When the last TLP packet is sent, an alarm is scheduled for the RTO period. When this alarm fires, the sender sends two packets, to evoke acknowledgements from the receiver, and restarts the RTO alarm.

Similar to TCP [RFC6298], the RTO period is set based on the following conditions:

- When the final TLP packet is sent, the RTO period is set to max(SRTT + 4*RTTVAR + MaxAckDelay, kMinRTOTimeout)

- When an RTO alarm fires, the RTO period is doubled.

The sender typically has incurred a high latency penalty by the time an RTO alarm fires, and this penalty increases exponentially in subsequent consecutive RTO events. Sending a single packet on an RTO event therefore makes the connection very sensitive to single packet loss. Sending two packets instead of one significantly increases resilience to packet drop in both directions, thus reducing the probability of consecutive RTO events.

QUIC's RTO algorithm differs from TCP in that the firing of an RTO alarm is not considered a strong enough signal of packet loss, so does not result in an immediate change to congestion window or recovery state. An RTO alarm fires only when there's a prolonged period of network silence, which could be caused by a change in the underlying network RTT.

QUIC also diverges from TCP by including MaxAckDelay in the RTO period. QUIC is able to explicitly model delay at the receiver via the ack delay field in the ACK frame. Since QUIC corrects for this delay in its SRTT and RTTVAR computations, it is necessary to add this delay explicitly in the TLP and RTO computation.

When an acknowledgment is received for a packet sent on an RTO event, any unacknowledged packets with lower packet numbers than those acknowledged MUST be marked as lost.

A packet sent when an RTO alarm fires MAY carry new data if available or unacknowledged data to potentially reduce recovery time. Since this packet is sent as a probe into the network prior to establishing any packet loss, prior unacknowledged packets SHOULD NOT be marked as lost.

A packet sent on an RTO alarm MUST NOT be blocked by the sender's congestion controller. A sender MUST however count these bytes as additional bytes in flight, since this packet adds network load without establishing packet loss.

3.4. Generating Acknowledgements

QUIC SHOULD delay sending acknowledgements in response to packets, but MUST NOT excessively delay acknowledgements of packets containing non-ack frames. Specifically, implementaions MUST attempt to enforce a maximum ack delay to avoid causing the peer spurious timeouts. The default maximum ack delay in QUIC is 25ms.

An acknowledgement MAY be sent for every second full-sized packet, as TCP does [RFC5681], or may be sent less frequently, as long as the delay does not exceed the maximum ack delay. QUIC recovery algorithms do not assume the peer generates an acknowledgement immediately when receiving a second full-sized packet.

Out-of-order packets SHOULD be acknowledged more quickly, in order to accelerate loss recovery. The receiver SHOULD send an immediate ACK when it receives a new packet which is not one greater than the largest received packet number.

As an optimization, a receiver MAY process multiple packets before sending any ACK frames in response. In this case they can determine whether an immediate or delayed acknowledgement should be generated after processing incoming packets.

3.4.1. ACK Ranges

When an ACK frame is sent, one or more ranges of acknowledged packets are included. Including older packets reduces the chance of spurious retransmits caused by losing previously sent ACK frames, at the cost of larger ACK frames.

ACK frames SHOULD always acknowledge the most recently received packets, and the more out-of-order the packets are, the more

important it is to send an updated ACK frame quickly, to prevent the peer from declaring a packet as lost and spuriously retransmitting the frames it contains.

Below is one recommended approach for determining what packets to include in an ACK frame.

3.4.2. Receiver Tracking of ACK Frames

When a packet containing an ACK frame is sent, the largest acknowledged in that frame may be saved. When a packet containing an ACK frame is acknowledged, the receiver can stop acknowledging packets less than or equal to the largest acknowledged in the sent ACK frame.

In cases without ACK frame loss, this algorithm allows for a minimum of 1 RTT of reordering. In cases with ACK frame loss, this approach does not guarantee that every acknowledgement is seen by the sender before it is no longer included in the ACK frame. Packets could be received out of order and all subsequent ACK frames containing them could be lost. In this case, the loss recovery algorithm may cause spurious retransmits, but the sender will continue making forward progress.

3.5. Pseudocode

3.5.1. Constants of interest

Constants used in loss recovery are based on a combination of RFCs, papers, and common practice. Some may need to be changed or negotiated in order to better suit a variety of environments.

kMaxTLPs (default 2): Maximum number of tail loss probes before an RTO fires.

kReorderingThreshold (default 3): Maximum reordering in packet number space before FACK style loss detection considers a packet lost.

kTimeReorderingFraction (default 1/8): Maximum reordering in time space before time based loss detection considers a packet lost. In fraction of an RTT.

kUsingTimeLossDetection (default false): Whether time based loss detection is in use. If false, uses FACK style loss detection.

kMinTLPTimeout (default 10ms): Minimum time in the future a tail loss probe alarm may be set for.

kMinRTOTimeout (default 200ms): Minimum time in the future an RTO alarm may be set for.

kDelayedAckTimeout (default 25ms): The length of the peer's delayed ack timer.

kDefaultInitialRtt (default 100ms): The default RTT used before an RTT sample is taken.

3.5.2. Variables of interest

Variables required to implement the congestion control mechanisms are described in this section.

loss_detection_alarm: Multi-modal alarm used for loss detection.

handshake_count: The number of times the handshake packets have been retransmitted without receiving an ack.

tlp_count: The number of times a tail loss probe has been sent without receiving an ack.

rto_count: The number of times an rto has been sent without receiving an ack.

largest_sent_before_rto: The last packet number sent prior to the first retransmission timeout.

time_of_last_sent_retransmittable_packet: The time the most recent retransmittable packet was sent.

time_of_last_sent_handshake_packet: The time the most recent packet containing handshake data was sent.

largest_sent_packet: The packet number of the most recently sent packet.

largest_acked_packet: The largest packet number acknowledged in an ACK frame.

latest_rtt: The most recent RTT measurement made when receiving an ack for a previously unacked packet.

smoothed_rtt: The smoothed RTT of the connection, computed as described in [RFC6298]

rttvar: The RTT variance, computed as described in [RFC6298]

min_rtt: The minimum RTT seen in the connection, ignoring ack delay.

max_ack_delay: The maximum ack delay in an incoming ACK frame for this connection. Excludes ack delays for ack only packets and those that create an RTT sample less than min_rtt.

reordering_threshold: The largest packet number gap between the largest acked retransmittable packet and an unacknowledged retransmittable packet before it is declared lost.

time_reordering_fraction: The reordering window as a fraction of max(smoothed_rtt, latest_rtt).

loss_time: The time at which the next packet will be considered lost based on early transmit or exceeding the reordering window in time.

sent_packets: An association of packet numbers to information about them, including a number field indicating the packet number, a time field indicating the time a packet was sent, a boolean indicating whether the packet is ack only, and a bytes field indicating the packet's size. sent_packets is ordered by packet number, and packets remain in sent_packets until acknowledged or lost.

3.5.3. Initialization

At the beginning of the connection, initialize the loss detection variables as follows:

```
  loss_detection_alarm.reset()
  handshake_count = 0
  tlp_count = 0
  rto_count = 0
  if (kUsingTimeLossDetection)
    reordering_threshold = infinite
    time_reordering_fraction = kTimeReorderingFraction
  else:
    reordering_threshold = kReorderingThreshold
    time_reordering_fraction = infinite
  loss_time = 0
  smoothed_rtt = 0
  rttvar = 0
  min_rtt = infinite
  max_ack_delay = 0
  largest_sent_before_rto = 0
  time_of_last_sent_retransmittable_packet = 0
  time_of_last_sent_handshake_packet = 0
  largest_sent_packet = 0
```

3.5.4. On Sending a Packet

After any packet is sent, be it a new transmission or a rebundled transmission, the following OnPacketSent function is called. The parameters to OnPacketSent are as follows:

- packet_number: The packet number of the sent packet.

- is_ack_only: A boolean that indicates whether a packet only contains an ACK frame. If true, it is still expected an ack will be received for this packet, but it is not retransmittable.

- is_handshake_packet: A boolean that indicates whether a packet contains handshake data.

- sent_bytes: The number of bytes sent in the packet, not including UDP or IP overhead, but including QUIC framing overhead.

Pseudocode for OnPacketSent follows:

```
    OnPacketSent(packet_number, is_ack_only, is_handshake_packet,
                 sent_bytes):
      largest_sent_packet = packet_number
      sent_packets[packet_number].packet_number = packet_number
      sent_packets[packet_number].time = now
      sent_packets[packet_number].ack_only = is_ack_only
      if !is_ack_only:
        if is_handshake_packet:
          time_of_last_sent_handshake_packet = now
        time_of_last_sent_retransmittable_packet = now
        OnPacketSentCC(sent_bytes)
        sent_packets[packet_number].bytes = sent_bytes
        SetLossDetectionAlarm()
```

3.5.5. On Ack Receipt

When an ack is received, it may acknowledge 0 or more packets.

Pseudocode for OnAckReceived and UpdateRtt follow:

```
OnAckReceived(ack):
  largest_acked_packet = ack.largest_acked
  // If the largest acked is newly acked, update the RTT.
  if (sent_packets[ack.largest_acked]):
    latest_rtt = now - sent_packets[ack.largest_acked].time
    UpdateRtt(latest_rtt, ack.ack_delay)
  // Find all newly acked packets.
  for acked_packet in DetermineNewlyAckedPackets():
    OnPacketAcked(acked_packet.packet_number)

  DetectLostPackets(ack.largest_acked_packet)
  SetLossDetectionAlarm()


UpdateRtt(latest_rtt, ack_delay):
  // min_rtt ignores ack delay.
  min_rtt = min(min_rtt, latest_rtt)
  // Adjust for ack delay if it's plausible.
  if (latest_rtt - min_rtt > ack_delay):
    latest_rtt -= ack_delay
    // Only save into max ack delay if it's used
    // for rtt calculation and is not ack only.
    if (!sent_packets[ack.largest_acked].ack_only)
      max_ack_delay = max(max_ack_delay, ack_delay)
  // Based on {{RFC6298}}.
  if (smoothed_rtt == 0):
    smoothed_rtt = latest_rtt
    rttvar = latest_rtt / 2
  else:
    rttvar_sample = abs(smoothed_rtt - latest_rtt)
    rttvar = 3/4 * rttvar + 1/4 * rttvar_sample
    smoothed_rtt = 7/8 * smoothed_rtt + 1/8 * latest_rtt
```

3.5.6. On Packet Acknowledgment

When a packet is acked for the first time, the following OnPacketAcked function is called. Note that a single ACK frame may newly acknowledge several packets. OnPacketAcked must be called once for each of these newly acked packets.

OnPacketAcked takes one parameter, acked_packet, which is the struct of the newly acked packet.

If this is the first acknowledgement following RTO, check if the smallest newly acknowledged packet is one sent by the RTO, and if so, inform congestion control of a verified RTO, similar to F-RTO [RFC5682]

Pseudocode for OnPacketAcked follows:

```
OnPacketAcked(acked_packet):
  if (!acked_packet.is_ack_only):
    OnPacketAckedCC(acked_packet)
  // If a packet sent prior to RTO was acked, then the RTO
  // was spurious.  Otherwise, inform congestion control.
  if (rto_count > 0 &&
      acked_packet.packet_number > largest_sent_before_rto)
    OnRetransmissionTimeoutVerified()
  handshake_count = 0
  tlp_count = 0
  rto_count = 0
  sent_packets.remove(acked_packet.packet_number)
```

3.5.7. Setting the Loss Detection Alarm

QUIC loss detection uses a single alarm for all timer-based loss detection. The duration of the alarm is based on the alarm's mode, which is set in the packet and timer events further below. The function SetLossDetectionAlarm defined below shows how the single timer is set based on the alarm mode.

3.5.7.1. Handshake Alarm

When a connection has unacknowledged handshake data, the handshake alarm is set and when it expires, all unacknowledgedd handshake data is retransmitted.

When stateless rejects are in use, the connection is considered immediately closed once a reject is sent, so no timer is set to retransmit the reject.

Version negotiation packets are always stateless, and MUST be sent once per handshake packet that uses an unsupported QUIC version, and MAY be sent in response to 0RTT packets.

3.5.7.2. Tail Loss Probe and Retransmission Alarm

Tail loss probes [TLP] and retransmission timeouts [RFC6298] are an alarm based mechanism to recover from cases when there are outstanding retransmittable packets, but an acknowledgement has not been received in a timely manner.

The TLP and RTO timers are armed when there is not unacknowledged handshake data. The TLP alarm is set until the max number of TLP packets have been sent, and then the RTO timer is set.

3.5.7.3. Early Retransmit Alarm

Early retransmit [RFC5827] is implemented with a 1/4 RTT timer. It is part of QUIC's time based loss detection, but is always enabled, even when only packet reordering loss detection is enabled.

3.5.7.4. Pseudocode

Pseudocode for SetLossDetectionAlarm follows:

```
 SetLossDetectionAlarm():
    // Don't arm the alarm if there are no packets with
    // retransmittable data in flight.
    if (bytes_in_flight == 0):
      loss_detection_alarm.cancel()
      return

    if (handshake packets are outstanding):
      // Handshake retransmission alarm.
      if (smoothed_rtt == 0):
        alarm_duration = 2 * kDefaultInitialRtt
      else:
        alarm_duration = 2 * smoothed_rtt
      alarm_duration = max(alarm_duration + max_ack_delay,
                           kMinTLPTimeout)
      alarm_duration = alarm_duration * (2 ^ handshake_count)
      loss_detection_alarm.set(
        time_of_last_sent_handshake_packet + alarm_duration)
      return;
    else if (loss_time != 0):
      // Early retransmit timer or time loss detection.
      alarm_duration = loss_time -
        time_of_last_sent_retransmittable_packet
    else:
      // RTO or TLP alarm
      // Calculate RTO duration
      alarm_duration =
        smoothed_rtt + 4 * rttvar + max_ack_delay
      alarm_duration = max(alarm_duration, kMinRTOTimeout)
      alarm_duration = alarm_duration * (2 ^ rto_count)
      if (tlp_count < kMaxTLPs):
        // Tail Loss Probe
        tlp_alarm_duration = max(1.5 * smoothed_rtt
                           + max_ack_delay, kMinTLPTimeout)
        alarm_duration = min(tlp_alarm_duration, alarm_duration)

    loss_detection_alarm.set(
      time_of_last_sent_retransmittable_packet + alarm_duration)
```

3.5.8. On Alarm Firing

QUIC uses one loss recovery alarm, which when set, can be in one of several modes. When the alarm fires, the mode determines the action to be performed.

Pseudocode for OnLossDetectionAlarm follows:

```
OnLossDetectionAlarm():
  if (handshake packets are outstanding):
    // Handshake retransmission alarm.
    RetransmitAllUnackedHandshakeData()
    handshake_count++
  else if (loss_time != 0):
    // Early retransmit or Time Loss Detection
    DetectLostPackets(largest_acked_packet)
  else if (tlp_count < kMaxTLPs):
    // Tail Loss Probe.
    SendOnePacket()
    tlp_count++
  else:
    // RTO.
    if (rto_count == 0)
      largest_sent_before_rto = largest_sent_packet
    SendTwoPackets()
    rto_count++

  SetLossDetectionAlarm()
```

3.5.9. Detecting Lost Packets

Packets in QUIC are only considered lost once a larger packet number is acknowledged. DetectLostPackets is called every time an ack is received. If the loss detection alarm fires and the loss_time is set, the previous largest acked packet is supplied.

3.5.9.1. Handshake Packets

The receiver MUST close the connection with an error of type OPTIMISTIC_ACK when receiving an unprotected packet that acks protected packets. The receiver MUST trust protected acks for unprotected packets, however. Aside from this, loss detection for handshake packets when an ack is processed is identical to other packets.

3.5.9.2. Pseudocode

DetectLostPackets takes one parameter, acked, which is the largest acked packet.

Pseudocode for DetectLostPackets follows:

```
DetectLostPackets(largest_acked):
  loss_time = 0
  lost_packets = {}
  delay_until_lost = infinite
  if (kUsingTimeLossDetection):
    delay_until_lost =
      (1 + time_reordering_fraction) *
          max(latest_rtt, smoothed_rtt)
  else if (largest_acked.packet_number == largest_sent_packet):
    // Early retransmit alarm.
    delay_until_lost = 5/4 * max(latest_rtt, smoothed_rtt)
  foreach (unacked < largest_acked.packet_number):
    time_since_sent = now() - unacked.time_sent
    delta = largest_acked.packet_number - unacked.packet_number
    if (time_since_sent > delay_until_lost ||
        delta > reordering_threshold):
      sent_packets.remove(unacked.packet_number)
      if (!unacked.is_ack_only):
        lost_packets.insert(unacked)
    else if (loss_time == 0 && delay_until_lost != infinite):
      loss_time = now() + delay_until_lost - time_since_sent

  // Inform the congestion controller of lost packets and
  // lets it decide whether to retransmit immediately.
  if (!lost_packets.empty()):
    OnPacketsLost(lost_packets)
```

3.6. Discussion

The majority of constants were derived from best common practices among widely deployed TCP implementations on the internet. Exceptions follow.

A shorter delayed ack time of 25ms was chosen because longer delayed acks can delay loss recovery and for the small number of connections where less than packet per 25ms is delivered, acking every packet is beneficial to congestion control and loss recovery.

The default initial RTT of 100ms was chosen because it is slightly higher than both the median and mean min_rtt typically observed on the public internet.

4. Congestion Control

QUIC's congestion control is based on TCP NewReno [RFC6582] congestion control to determine the congestion window. QUIC congestion control is specified in bytes due to finer control and the ease of appropriate byte counting [RFC3465].

QUIC hosts MUST NOT send packets if they would increase bytes_in_flight (defined in Section 4.7.2) beyond the available congestion window, unless the packet is a probe packet sent after the TLP or RTO alarm fires, as described in Section 3.3.2 and Section 3.3.3.

4.1. Slow Start

QUIC begins every connection in slow start and exits slow start upon loss. QUIC re-enters slow start anytime the congestion window is less than sshthresh, which typically only occurs after an RTO. While in slow start, QUIC increases the congestion window by the number of acknowledged bytes when each ack is processed.

4.2. Congestion Avoidance

Slow start exits to congestion avoidance. Congestion avoidance in NewReno uses an additive increase multiplicative decrease (AIMD) approach that increases the congestion window by one MSS of bytes per congestion window acknowledged. When a loss is detected, NewReno halves the congestion window and sets the slow start threshold to the new congestion window.

4.3. Recovery Period

Recovery is a period of time beginning with detection of a lost packet. Because QUIC retransmits stream data and control frames, not packets, it defines the end of recovery as a packet sent after the start of recovery being acknowledged. This is slightly different from TCP's definition of recovery ending when the lost packet that started recovery is acknowledged.

During recovery, the congestion window is not increased or decreased. As such, multiple lost packets only decrease the congestion window once as long as they're lost before exiting recovery. This causes QUIC to decrease the congestion window multiple times if retransmisions are lost, but limits the reduction to once per round trip.

4.4. Tail Loss Probe

A TLP packet MUST NOT be blocked by the sender's congestion controller. The sender MUST however count these bytes as additional bytes-in-flight, since a TLP adds network load without establishing packet loss.

Acknowledgement or loss of tail loss probes are treated like any other packet.

4.5. Retransmission Timeout

When retransmissions are sent due to a retransmission timeout alarm, no change is made to the congestion window until the next acknowledgement arrives. The retransmission timeout is considered spurious when this acknowledgement acknowledges packets sent prior to the first retransmission timeout. The retransmission timeout is considered valid when this acknowledgement acknowledges no packets sent prior to the first retransmission timeout. In this case, the congestion window MUST be reduced to the minimum congestion window and slow start is re-entered.

4.6. Pacing

This document does not specify a pacer, but it is RECOMMENDED that a sender pace sending of all retransmittable packets based on input from the congestion controller. For example, a pacer might distribute the congestion window over the SRTT when used with a window-based controller, and a pacer might use the rate estimate of a rate-based controller.

An implementation should take care to architect its congestion controller to work well with a pacer. For instance, a pacer might wrap the congestion controller and control the availability of the congestion window, or a pacer might pace out packets handed to it by the congestion controller. Timely delivery of ACK frames is important for efficient loss recovery. Packets containing only ACK frames should therefore not be paced, to avoid delaying their delivery to the peer.

As an example of a well-known and publicly available implementation of a flow pacer, implementers are referred to the Fair Queue packet scheduler (fq qdisc) in Linux (3.11 onwards).

4.7. Pseudocode

4.7.1. Constants of interest

Constants used in congestion control are based on a combination of RFCs, papers, and common practice. Some may need to be changed or negotiated in order to better suit a variety of environments.

kDefaultMss (default 1460 bytes): The default max packet size used for calculating default and minimum congestion windows.

kInitialWindow (default 10 * kDefaultMss): Default limit on the amount of outstanding data in bytes.

kMinimumWindow (default 2 * kDefaultMss): Default minimum congestion window.

kLossReductionFactor (default 0.5): Reduction in congestion window when a new loss event is detected.

4.7.2. Variables of interest

Variables required to implement the congestion control mechanisms are described in this section.

bytes_in_flight: The sum of the size in bytes of all sent packets that contain at least one retransmittable frame, and have not been acked or declared lost. The size does not include IP or UDP overhead. Packets only containing ACK frames do not count towards bytes_in_flight to ensure congestion control does not impede congestion feedback.

congestion_window: Maximum number of bytes-in-flight that may be sent.

end_of_recovery: The largest packet number sent when QUIC detects a loss. When a larger packet is acknowledged, QUIC exits recovery.

ssthresh: Slow start threshold in bytes. When the congestion window is below ssthresh, the mode is slow start and the window grows by the number of bytes acknowledged.

4.7.3. Initialization

At the beginning of the connection, initialize the congestion control variables as follows:

```
congestion_window = kInitialWindow
bytes_in_flight = 0
end_of_recovery = 0
ssthresh = infinite
```

4.7.4. On Packet Sent

Whenever a packet is sent, and it contains non-ACK frames, the packet increases bytes_in_flight.

```
OnPacketSentCC(bytes_sent):
  bytes_in_flight += bytes_sent
```

4.7.5. On Packet Acknowledgement

Invoked from loss detection's OnPacketAcked and is supplied with acked_packet from sent_packets.

```
InRecovery(packet_number)
  return packet_number <= end_of_recovery

OnPacketAckedCC(acked_packet):
  // Remove from bytes_in_flight.
  bytes_in_flight -= acked_packet.bytes
  if (InRecovery(acked_packet.packet_number)):
    // Do not increase congestion window in recovery period.
    return
  if (congestion_window < ssthresh):
    // Slow start.
    congestion_window += acked_packet.bytes
  else:
    // Congestion avoidance.
    congestion_window +=
      kDefaultMss * acked_packet.bytes / congestion_window
```

4.7.6. On Packets Lost

Invoked by loss detection from DetectLostPackets when new packets are detected lost.

```
    OnPacketsLost(lost_packets):
      // Remove lost packets from bytes_in_flight.
      for (lost_packet : lost_packets):
        bytes_in_flight -= lost_packet.bytes
      largest_lost_packet = lost_packets.last()
      // Start a new recovery epoch if the lost packet is larger
      // than the end of the previous recovery epoch.
      if (!InRecovery(largest_lost_packet.packet_number)):
        end_of_recovery = largest_sent_packet
        congestion_window *= kLossReductionFactor
        congestion_window = max(congestion_window, kMinimumWindow)
        ssthresh = congestion_window
```

4.7.7. On Retransmission Timeout Verified

QUIC decreases the congestion window to the minimum value once the retransmission timeout has been verified.

```
    OnRetransmissionTimeoutVerified()
      congestion_window = kMinimumWindow
```

5. IANA Considerations

This document has no IANA actions. Yet.

6. References

6.1. Normative References

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-11 (work in progress), April 2018.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

6.2. Informative References

[RFC3465] Allman, M., "TCP Congestion Control with Appropriate Byte Counting (ABC)", RFC 3465, DOI 10.17487/RFC3465, February 2003, <https://www.rfc-editor.org/info/rfc3465>.

[RFC4653] Bhandarkar, S., Reddy, A., Allman, M., and E. Blanton, "Improving the Robustness of TCP to Non-Congestion Events", RFC 4653, DOI 10.17487/RFC4653, August 2006, <https://www.rfc-editor.org/info/rfc4653>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/info/rfc5681>.

[RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, DOI 10.17487/RFC5682, September 2009, <https://www.rfc-editor.org/info/rfc5682>.

[RFC5827] Allman, M., Avrachenkov, K., Ayesta, U., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, DOI 10.17487/RFC5827, May 2010, <https://www.rfc-editor.org/info/rfc5827>.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <https://www.rfc-editor.org/info/rfc6298>.

[RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, DOI 10.17487/RFC6582, April 2012, <https://www.rfc-editor.org/info/rfc6582>.

[RFC6675] Blanton, E., Allman, M., Wang, L., Jarvinen, I., Kojo, M., and Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP", RFC 6675, DOI 10.17487/RFC6675, August 2012, <https://www.rfc-editor.org/info/rfc6675>.

[TLP] Dukkipati, N., Cardwell, N., Cheng, Y., and M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses", draft-dukkipati-tcpm-tcp-loss-probe-01 (work in progress), February 2013.

6.3. URIs

[1] https://mailarchive.ietf.org/arch/search/?email_list=quic

[2] https://github.com/quicwg

[3] https://github.com/quicwg/base-drafts/labels/-recovery

Appendix A. Acknowledgments

Appendix B. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

B.1. Since draft-ietf-quic-recovery-10

- Improved text on ack generation (#1139, #1159)
- Make references to TCP recovery mechanisms informational (#1195)
- Define time_of_last_sent_handshake_packet (#1171)
- Added signal from TLS the data it includes needs to be sent in a Retry packet (#1061, #1199)
- Minimum RTT (min_rtt) is initialized with an infinite value (#1169)

B.2. Since draft-ietf-quic-recovery-09

No significant changes.

B.3. Since draft-ietf-quic-recovery-08

- Clarified pacing and RTO (#967, #977)

B.4. Since draft-ietf-quic-recovery-07

- Include Ack Delay in RTO(and TLP) computations (#981)
- Ack Delay in SRTT computation (#961)
- Default RTT and Slow Start (#590)
- Many editorial fixes.

B.5. Since draft-ietf-quic-recovery-06

No significant changes.

B.6. Since draft-ietf-quic-recovery-05

- Add more congestion control text (#776)

B.7. Since draft-ietf-quic-recovery-04

No significant changes.

B.8. Since draft-ietf-quic-recovery-03

No significant changes.

B.9. Since draft-ietf-quic-recovery-02

- Integrate F-RTO (#544, #409)
- Add congestion control (#545, #395)
- Require connection abort if a skipped packet was acknowledged (#415)
- Simplify RTO calculations (#142, #417)

B.10. Since draft-ietf-quic-recovery-01

- Overview added to loss detection
- Changes initial default RTT to 100ms
- Added time-based loss detection and fixes early retransmit
- Clarified loss recovery for handshake packets
- Fixed references and made TCP references informative

B.11. Since draft-ietf-quic-recovery-00

- Improved description of constants and ACK behavior

B.12. Since draft-iyengar-quic-loss-recovery-01

- Adopted as base for draft-ietf-quic-recovery
- Updated authors/editors list
- Added table of contents

Authors' Addresses

Jana Iyengar (editor)
Fastly

Email: jri.ietf@gmail.com

Ian Swett (editor)
Google

Email: ianswett@google.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

M. Thomson, Ed.
Mozilla
S. Turner, Ed.
sn3rd

March 13, 2017

Using Transport Layer Security (TLS) to Secure QUIC
draft-ietf-quic-tls-02

Abstract

This document describes how Transport Layer Security (TLS) can be used to secure QUIC.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic .

Working Group information can be found at https://github.com/quicwg ; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/tls .

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 14, 2017.

Copyright Notice

Table of Contents

1. Introduction

QUIC [QUIC-TRANSPORT] provides a multiplexed transport. When used for HTTP [RFC7230] semantics [QUIC-HTTP] it provides several key advantages over HTTP/1.1 [RFC7230] or HTTP/2 [RFC7540] over TCP [RFC0793].

This document describes how QUIC can be secured using Transport Layer Security (TLS) version 1.3 [I-D.ietf-tls-tls13]. TLS 1.3 provides critical latency improvements for connection establishment over previous versions. Absent packet loss, most new connections can be established and secured within a single round trip; on subsequent connections between the same client and server, the client can often send application data immediately, that is, using a zero round trip setup.

This document describes how the standardized TLS 1.3 can act a security component of QUIC. The same design could work for TLS 1.2, though few of the benefits QUIC provides would be realized due to the handshake latency in versions of TLS prior to 1.3.

2. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

This document uses the terminology established in [QUIC-TRANSPORT].

For brevity, the acronym TLS is used to refer to TLS 1.3.

TLS terminology is used when referring to parts of TLS. Though TLS assumes a continuous stream of octets, it divides that stream into _records_. Most relevant to QUIC are the records that contain TLS _handshake messages_, which are discrete messages that are used for key agreement, authentication and parameter negotiation. Ordinarily, TLS records can also contain _application data_, though in the QUIC usage there is no use of TLS application data.

3. Protocol Overview

QUIC [QUIC-TRANSPORT] assumes responsibility for the confidentiality and integrity protection of packets. For this it uses keys derived from a TLS 1.3 connection [I-D.ietf-tls-tls13]; QUIC also relies on TLS 1.3 for authentication and negotiation of parameters that are critical to security and performance.

Rather than a strict layering, these two protocols are co-dependent: QUIC uses the TLS handshake; TLS uses the reliability and ordered delivery provided by QUIC streams.

This document defines how QUIC interacts with TLS. This includes a description of how TLS is used, how keying material is derived from TLS, and the application of that keying material to protect QUIC packets. Figure 1 shows the basic interactions between TLS and QUIC, with the QUIC packet protection being called out specially.

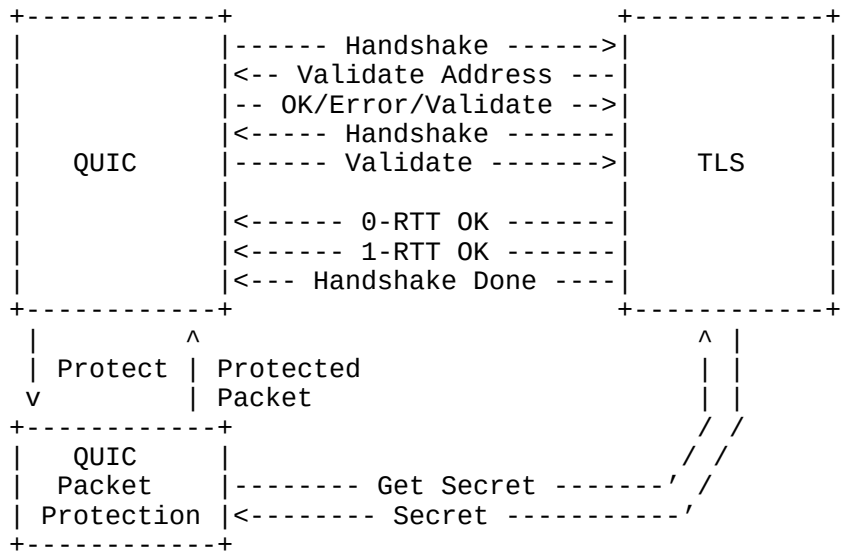


Figure 1: QUIC and TLS Interactions

The initial state of a QUIC connection has packets exchanged without any form of protection. In this state, QUIC is limited to using stream 1 and associated packets. Stream 1 is reserved for a TLS connection. This is a complete TLS connection as it would appear when layered over TCP; the only difference is that QUIC provides the reliability and ordering that would otherwise be provided by TCP.

At certain points during the TLS handshake, keying material is exported from the TLS connection for use by QUIC. This keying material is used to derive packet protection keys. Details on how and when keys are derived and used are included in Section 5.

This arrangement means that some TLS messages receive redundant protection from both the QUIC packet protection and the TLS record protection. These messages are limited in number; the TLS connection is rarely needed once the handshake completes.

3.1. TLS Overview

TLS provides two endpoints a way to establish a means of communication over an untrusted medium (that is, the Internet) that ensures that messages they exchange cannot be observed, modified, or forged.

TLS features can be separated into two basic functions: an authenticated key exchange and record protection. QUIC primarily

uses the authenticated key exchange provided by TLS but provides its own packet protection.

The TLS authenticated key exchange occurs between two entities: client and server. The client initiates the exchange and the server responds. If the key exchange completes successfully, both client and server will agree on a secret. TLS supports both pre-shared key (PSK) and Diffie-Hellman (DH) key exchanges. PSK is the basis for 0-RTT; the latter provides perfect forward secrecy (PFS) when the DH keys are destroyed.

After completing the TLS handshake, the client will have learned and authenticated an identity for the server and the server is optionally able to learn and authenticate an identity for the client. TLS supports X.509 certificate-based authentication [RFC5280] for both server and client.

The TLS key exchange is resistent to tampering by attackers and it produces shared secrets that cannot be controlled by either participating peer.

3.2. TLS Handshake

TLS 1.3 provides two basic handshake modes of interest to QUIC:

- A full, 1-RTT handshake in which the client is able to send application data after one round trip and the server immediately after receiving the first handshake message from the client.

- A 0-RTT handshake in which the client uses information it has previously learned about the server to send immediately. This data can be replayed by an attacker so it MUST NOT carry a self-contained trigger for any non-idempotent action.

A simplified TLS 1.3 handshake with 0-RTT application data is shown in Figure 2, see [I-D.ietf-tls-tls13] for more options and details.

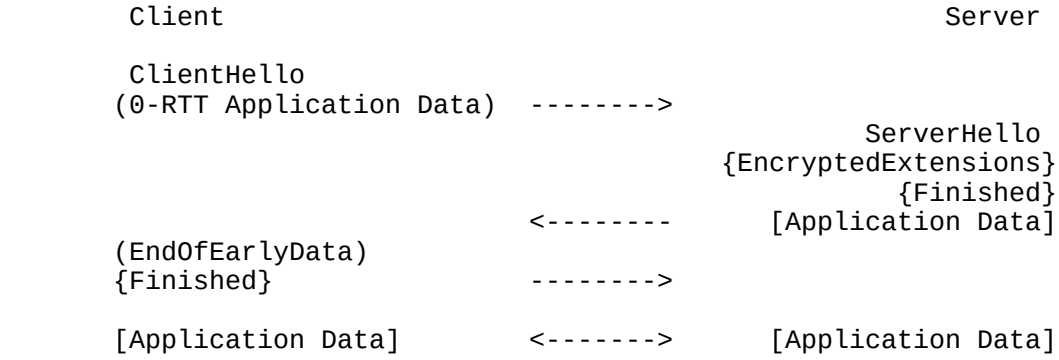


Figure 2: TLS Handshake with 0-RTT

This 0-RTT handshake is only possible if the client and server have previously communicated. In the 1-RTT handshake, the client is unable to send protected application data until it has received all of the handshake messages sent by the server.

Two additional variations on this basic handshake exchange are relevant to this document:

- The server can respond to a ClientHello with a HelloRetryRequest, which adds an additional round trip prior to the basic exchange. This is needed if the server wishes to request a different key exchange key from the client. HelloRetryRequest is also used to verify that the client is correctly able to receive packets on the address it claims to have (see [QUIC-TRANSPORT]).

- A pre-shared key mode can be used for subsequent handshakes to avoid public key operations. This is the basis for 0-RTT data, even if the remainder of the connection is protected by a new Diffie-Hellman exchange.

4. TLS Usage

QUIC reserves stream 1 for a TLS connection. Stream 1 contains a complete TLS connection, which includes the TLS record layer. Other than the definition of a QUIC-specific extension (see Section-TBD), TLS is unmodified for this use. This means that TLS will apply confidentiality and integrity protection to its records. In particular, TLS record protection is what provides confidentiality protection for the TLS handshake messages sent by the server.

QUIC permits a client to send frames on streams starting from the first packet. The initial packet from a client contains a stream frame for stream 1 that contains the first TLS handshake messages

from the client. This allows the TLS handshake to start with the first packet that a client sends.

QUIC packets are protected using a scheme that is specific to QUIC, see Section 5. Keys are exported from the TLS connection when they become available using a TLS exporter (see Section 7.3.3 of [I-D.ietf-tls-tls13] and Section 5.2). After keys are exported from TLS, QUIC manages its own key schedule.

4.1. Handshake and Setup Sequence

The integration of QUIC with a TLS handshake is shown in more detail in Figure 3. QUIC "STREAM" frames on stream 1 carry the TLS handshake. QUIC performs loss recovery [QUIC-RECOVERY] for this stream and ensures that TLS handshake messages are delivered in the correct order.

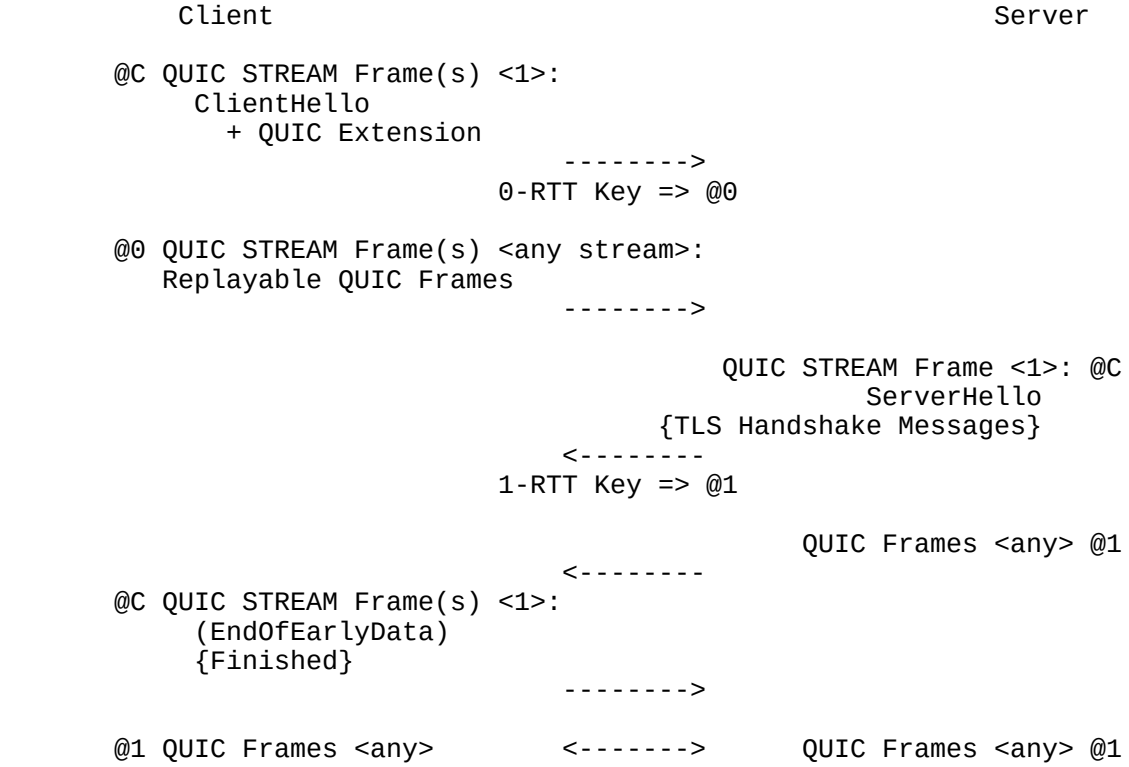


Figure 3: QUIC over TLS Handshake

In Figure 3, symbols mean:

- "<" and ">" enclose stream numbers.
- "@" indicates the key phase that is currently used for protecting QUIC packets.
- "(" and ")" enclose messages that are protected with TLS 0-RTT handshake or application keys.
- "{" and "}" enclose messages that are protected by the TLS Handshake keys.

If 0-RTT is not attempted, then the client does not send packets protected by the 0-RTT key (@0). In that case, the only key transition on the client is from unprotected packets (@C) to 1-RTT protection (@1), which happens after it sends its final set of TLS handshake messages.

The server sends TLS handshake messages without protection (@C). The server transitions from no protection (@C) to full 1-RTT protection (@1) after it sends the last of its handshake messages.

Some TLS handshake messages are protected by the TLS handshake record protection. These keys are not exported from the TLS connection for use in QUIC. QUIC packets from the server are sent in the clear until the final transition to 1-RTT keys.

The client transitions from cleartext (@C) to 0-RTT keys (@0) when sending 0-RTT data, and subsequently to to 1-RTT keys (@1) after its second flight of TLS handshake messages. This creates the potential for unprotected packets to be received by a server in close proximity to packets that are protected with 1-RTT keys.

More information on key transitions is included in Section 6.1.

4.2. Interface to TLS

As shown in Figure 1, the interface from QUIC to TLS consists of four primary functions: Handshake, Source Address Validation, Key Ready Events, and Secret Export.

Additional functions might be needed to configure TLS.

4.2.1. Handshake Interface

In order to drive the handshake, TLS depends on being able to send and receive handshake messages on stream 1. There are two basic functions on this interface: one where QUIC requests handshake messages and one where QUIC provides handshake packets.

Before starting the handshake QUIC provides TLS with the transport parameters (see Section 9.2) that it wishes to carry.

A QUIC client starts TLS by requesting TLS handshake octets from TLS. The client acquires handshake octets before sending its first packet.

A QUIC server starts the process by providing TLS with stream 1 octets.

Each time that an endpoint receives data on stream 1, it delivers the octets to TLS if it is able. Each time that TLS is provided with new data, new handshake octets are requested from TLS. TLS might not provide any octets if the handshake messages it has received are incomplete or it has no data to send.

Once the TLS handshake is complete, this is indicated to QUIC along with any final handshake octets that TLS needs to send. TLS also provides QUIC with the transport parameters that the peer advertised during the handshake.

Once the handshake is complete, TLS becomes passive. TLS can still receive data from its peer and respond in kind, but it will not need to send more data unless specifically requested - either by an application or QUIC. One reason to send data is that the server might wish to provide additional or updated session tickets to a client.

When the handshake is complete, QUIC only needs to provide TLS with any data that arrives on stream 1. In the same way that is done during the handshake, new data is requested from TLS after providing received data.

Important: Until the handshake is reported as complete, the connection and key exchange are not properly authenticated at the server. Even though 1-RTT keys are available to a server after receiving the first handshake messages from a client, the server cannot consider the client to be authenticated until it receives and validates the client’s Finished message.

The requirement for the server to wait for the client Finished message creates a dependency on that message being delivered. A client can avoid the potential for head-of-line blocking that this implies by sending a copy of the STREAM frame that carries the Finished message in multiple packets. This enables immediate server processing for those packets.

4.2.2. Source Address Validation

During the processing of the TLS ClientHello, TLS requests that the transport make a decision about whether to request source address validation from the client.

An initial TLS ClientHello that resumes a session includes an address validation token in the session ticket; this includes all attempts at 0-RTT. If the client does not attempt session resumption, no token will be present. While processing the initial ClientHello, TLS provides QUIC with any token that is present. In response, QUIC provides one of three responses:

- proceed with the connection,
- ask for client address validation, or
- abort the connection.

If QUIC requests source address validation, it also provides a new address validation token. TLS includes that along with any information it requires in the cookie extension of a TLS HelloRetryRequest message. In the other cases, the connection either proceeds or terminates with a handshake error.

The client echoes the cookie extension in a second ClientHello. A ClientHello that contains a valid cookie extension will be always be in response to a HelloRetryRequest. If address validation was requested by QUIC, then this will include an address validation token. TLS makes a second address validation request of QUIC, including the value extracted from the cookie extension. In response to this request, QUIC cannot ask for client address validation, it can only abort or permit the connection attempt to proceed.

QUIC can provide a new address validation token for use in session resumption at any time after the handshake is complete. Each time a new token is provided TLS generates a NewSessionTicket message, with the token included in the ticket.

See Section 7 for more details on client address validation.

4.2.3. Key Ready Events

TLS provides QUIC with signals when 0-RTT and 1-RTT keys are ready for use. These events are not asynchronous, they always occur immediately after TLS is provided with new handshake octets, or after TLS produces handshake octets.

When TLS completed its handshake, 1-RTT keys can be provided to QUIC. On both client and server, this occurs after sending the TLS Finished message.

This ordering means that there could be frames that carry TLS handshake messages ready to send at the same time that application data is available. An implementation MUST ensure that TLS handshake messages are always sent in cleartext packets. Separate packets are required for data that needs protection from 1-RTT keys.

If 0-RTT is possible, it is ready after the client sends a TLS ClientHello message or the server receives that message. After providing a QUIC client with the first handshake octets, the TLS stack might signal that 0-RTT keys are ready. On the server, after receiving handshake octets that contain a ClientHello message, a TLS server might signal that 0-RTT keys are available.

1-RTT keys are used for packets in both directions. 0-RTT keys are only used to protect packets sent by the client.

4.2.4. Secret Export

Details how secrets are exported from TLS are included in Section 5.2.

4.2.5. TLS Interface Summary

Figure 4 summarizes the exchange between QUIC and TLS for both client and server.

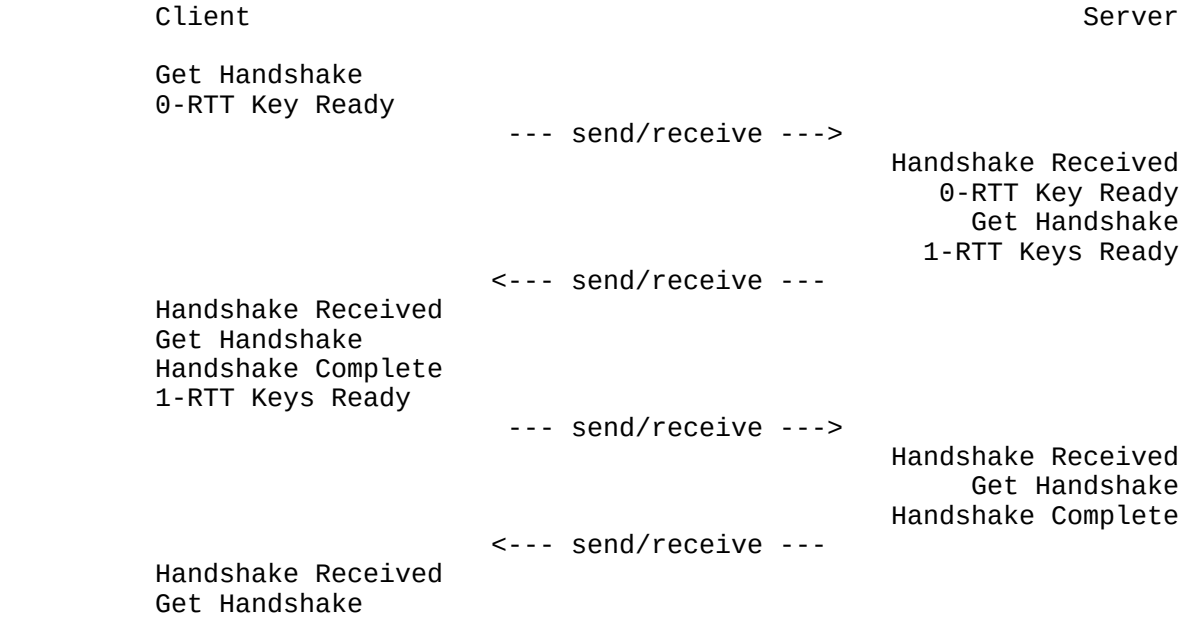


Figure 4: Interaction Summary between QUIC and TLS

4.3. TLS Version

This document describes how TLS 1.3 [I-D.ietf-tls-tls13] is used with QUIC.

In practice, the TLS handshake will negotiate a version of TLS to use. This could result in a newer version of TLS than 1.3 being negotiated if both endpoints support that version. This is acceptable provided that the features of TLS 1.3 that are used by QUIC are supported by the newer version.

A badly configured TLS implementation could negotiate TLS 1.2 or another older version of TLS. An endpoint MUST terminate the connection if a version of TLS older than 1.3 is negotiated.

4.4. ClientHello Size

QUIC requires that the initial handshake packet from a client fit within a single packet of at least 1280 octets. With framing and packet overheads this value could be reduced.

A TLS ClientHello can fit within this limit with ample space remaining. However, there are several variables that could cause this limit to be exceeded. Implementations are reminded that large

session tickets or HelloRetryRequest cookies, multiple or large key shares, and long lists of supported ciphers, signature algorithms, versions, QUIC transport parameters, and other negotiable parameters and extensions could cause this message to grow.

For servers, the size of the session tickets and HelloRetryRequest cookie extension can have an effect on a client's ability to connect. Choosing a small value increases the probability that these values can be successfully used by a client.

A TLS implementation does not need to enforce this size constraint. QUIC padding can be used to reach this size, meaning that a TLS server is unlikely to receive a large ClientHello message.

4.5. Peer Authentication

The requirements for authentication depend on the application protocol that is in use. TLS provides server authentication and permits the server to request client authentication.

A client MUST authenticate the identity of the server. This typically involves verification that the identity of the server is included in a certificate and that the certificate is issued by a trusted entity (see for example [RFC2818]).

A server MAY request that the client authenticate during the handshake. A server MAY refuse a connection if the client is unable to authenticate when requested. The requirements for client authentication vary based on application protocol and deployment.

A server MUST NOT use post-handshake client authentication (see Section 4.6.2 of [I-D.ietf-tls-tls13]).

4.6. TLS Errors

Errors in the TLS connection SHOULD be signaled using TLS alerts on stream 1. A failure in the handshake MUST be treated as a QUIC connection error of type TLS_HANDSHAKE_FAILED. Once the handshake is complete, an error in the TLS connection that causes a TLS alert to be sent or received MUST be treated as a QUIC connection error of type TLS_FATAL_ALERT_GENERATED or TLS_FATAL_ALERT_RECEIVED respectively.

5. QUIC Packet Protection

QUIC packet protection provides authenticated encryption of packets. This provides confidentiality and integrity protection for the

content of packets (see Section 5.3). Packet protection uses keys that are exported from the TLS connection (see Section 5.2).

Different keys are used for QUIC packet protection and TLS record protection. Having separate QUIC and TLS record protection means that TLS records can be protected by two different keys. This redundancy is limited to only a few TLS records, and is maintained for the sake of simplicity.

5.1. Installing New Keys

As TLS reports the availability of keying material, the packet protection keys and initialization vectors (IVs) are updated (see Section 5.2). The selection of AEAD function is also updated to match the AEAD negotiated by TLS.

For packets other than any unprotected handshake packets (see Section 6.1), once a change of keys has been made, packets with higher packet numbers MUST use the new keying material. The KEY_PHASE bit on these packets is inverted each time new keys are installed to signal the use of the new keys to the recipient (see Section 6 for details).

An endpoint retransmits stream data in a new packet. New packets have new packet numbers and use the latest packet protection keys. This simplifies key management when there are key updates (see Section 6.2).

5.2. QUIC Key Expansion

QUIC uses a system of packet protection secrets, keys and IVs that are modelled on the system used in TLS [I-D.ietf-tls-tls13]. The secrets that QUIC uses as the basis of its key schedule are obtained using TLS exporters (see Section 7.3.3 of [I-D.ietf-tls-tls13]).

QUIC uses HKDF with the same hash function negotiated by TLS for key derivation. For example, if TLS is using the TLS_AES_128_GCM_SHA256, the SHA-256 hash function is used.

5.2.1. 0-RTT Secret

0-RTT keys are those keys that are used in resumed connections prior to the completion of the TLS handshake. Data sent using 0-RTT keys might be replayed and so has some restrictions on its use, see Section 8.2. 0-RTT keys are used after sending or receiving a ClientHello.

The secret is exported from TLS using the exporter label "EXPORTER-QUIC 0-RTT Secret" and an empty context. The size of the secret MUST be the size of the hash output for the PRF hash function negotiated by TLS. This uses the TLS early_exporter_secret. The QUIC 0-RTT secret is only used for protection of packets sent by the client.

```
client_0rtt_secret
    = TLS-Exporter("EXPORTER-QUIC 0-RTT Secret"
                   "", Hash.length)
```

5.2.2. 1-RTT Secrets

1-RTT keys are used by both client and server after the TLS handshake completes. There are two secrets used at any time: one is used to derive packet protection keys for packets sent by the client, the other for packet protection keys on packets sent by the server.

The initial client packet protection secret is exported from TLS using the exporter label "EXPORTER-QUIC client 1-RTT Secret"; the initial server packet protection secret uses the exporter label "EXPORTER-QUIC server 1-RTT Secret". Both exporters use an empty context. The size of the secret MUST be the size of the hash output for the PRF hash function negotiated by TLS.

```
client_pp_secret_0
    = TLS-Exporter("EXPORTER-QUIC client 1-RTT Secret"
                   "", Hash.length)
server_pp_secret_0
    = TLS-Exporter("EXPORTER-QUIC server 1-RTT Secret"
                   "", Hash.length)
```

These secrets are used to derive the initial client and server packet protection keys.

After a key update (see Section 6.2), these secrets are updated using the HKDF-Expand-Label function defined in Section 7.1 of [I-D.ietf-tls-tls13]. HKDF-Expand-Label uses the PRF hash function negotiated by TLS. The replacement secret is derived using the existing Secret, a Label of "QUIC client 1-RTT Secret" for the client and "QUIC server 1-RTT Secret" for the server, an empty HashValue, and the same output Length as the hash function selected by TLS for its PRF.

```
client_pp_secret_<N+1>
    = HKDF-Expand-Label(client_pp_secret_<N>,
                        "QUIC client 1-RTT Secret",
                        "", Hash.length)
server_pp_secret_<N+1>
    = HKDF-Expand-Label(server_pp_secret_<N>,
                        "QUIC server 1-RTT Secret",
                        "", Hash.length)
```

This allows for a succession of new secrets to be created as needed.

HKDF-Expand-Label uses HKDF-Expand [RFC5869] with a specially formatted info parameter. The info parameter that includes the output length (in this case, the size of the PRF hash output) encoded on two octets in network byte order, the length of the prefixed Label as a single octet, the value of the Label prefixed with "TLS 1.3, ", and a zero octet to indicate an empty HashValue. For example, the client packet protection secret uses an info parameter of:

```
info = (HashLen / 256) || (HashLen % 256) || 0x21 ||
       "TLS 1.3, QUIC client 1-RTT secret" || 0x00
```

5.2.3. Packet Protection Key and IV

The complete key expansion uses an identical process for key expansion as defined in Section 7.3 of [I-D.ietf-tls-tls13], using different values for the input secret. QUIC uses the AEAD function negotiated by TLS.

The packet protection key and IV used to protect the 0-RTT packets sent by a client use the QUIC 0-RTT secret. This uses the HKDF-Expand-Label with the PRF hash function negotiated by TLS.

The length of the output is determined by the requirements of the AEAD function selected by TLS. The key length is the AEAD key size. As defined in Section 5.3 of [I-D.ietf-tls-tls13], the IV length is the larger of 8 or N_MIN (see Section 4 of [RFC5116]).

```
client_0rtt_key = HKDF-Expand-Label(client_0rtt_secret,
                                    "key", "", key_length)
client_0rtt_iv = HKDF-Expand-Label(client_0rtt_secret,
                                   "iv", "", iv_length)
```

Similarly, the packet protection key and IV used to protect 1-RTT packets sent by both client and server use the current packet protection secret.

```
client_pp_key_<N> = HKDF-Expand-Label(client_pp_secret_<N>,
                                      "key", "", key_length)
client_pp_iv_<N> = HKDF-Expand-Label(client_pp_secret_<N>,
                                     "iv", "", iv_length)
server_pp_key_<N> = HKDF-Expand-Label(server_pp_secret_<N>,
                                      "key", "", key_length)
server_pp_iv_<N> = HKDF-Expand-Label(server_pp_secret_<N>,
                                     "iv", "", iv_length)
```

The client protects (or encrypts) packets with the client packet protection key and IV; the server protects packets with the server packet protection key.

The QUIC record protection initially starts without keying material. When the TLS state machine reports that the ClientHello has been sent, the 0-RTT keys can be generated and installed for writing. When the TLS state machine reports completion of the handshake, the 1-RTT keys can be generated and installed for writing.

5.3. QUIC AEAD Usage

The Authentication Encryption with Associated Data (AEAD) [RFC5116] function used for QUIC packet protection is AEAD that is negotiated for use with the TLS connection. For example, if TLS is using the TLS_AES_128_GCM_SHA256, the AEAD_AES_128_GCM function is used.

Regular QUIC packets are protected by an AEAD [RFC5116]. Version negotiation and public reset packets are not protected.

Once TLS has provided a key, the contents of regular QUIC packets immediately after any TLS messages have been sent are protected by the AEAD selected by TLS.

The key, K, for the AEAD is either the client packet protection key (client_pp_key_n) or the server packet protection key (server_pp_key_n), derived as defined in Section 5.2.

The nonce, N, for the AEAD is formed by combining either the packet protection IV (either client_pp_iv_n or server_pp_iv_n) with packet numbers. The 64 bits of the reconstructed QUIC packet number in network byte order is left-padded with zeros to the size of the IV. The exclusive OR of the padded packet number and the IV forms the AEAD nonce.

The associated data, A, for the AEAD is the contents of the QUIC header, starting from the flags octet in the common header.

The input plaintext, P, for the AEAD is the contents of the QUIC frame following the packet number, as described in [QUIC-TRANSPORT].

The output ciphertext, C, of the AEAD is transmitted in place of P.

Prior to TLS providing keys, no record protection is performed and the plaintext, P, is transmitted unmodified.

5.4. Packet Numbers

QUIC has a single, contiguous packet number space. In comparison, TLS restarts its sequence number each time that record protection keys are changed. The sequence number restart in TLS ensures that a compromise of the current traffic keys does not allow an attacker to truncate the data that is sent after a key update by sending additional packets under the old key (causing new packets to be discarded).

QUIC does not assume a reliable transport and is required to handle attacks where packets are dropped in other ways. QUIC is therefore not affected by this form of truncation.

The QUIC packet number is not reset and it is not permitted to go higher than its maximum value of 2^64-1. This establishes a hard limit on the number of packets that can be sent.

Some AEAD functions have limits for how many packets can be encrypted under the same key and IV (see for example [AEBounds]). This might be lower than the packet number limit. An endpoint MUST initiate a key update (Section 6.2) prior to exceeding any limit set for the AEAD that is in use.

TLS maintains a separate sequence number that is used for record protection on the connection that is hosted on stream 1. This sequence number is not visible to QUIC.

5.5. Receiving Protected Packets

Once an endpoint successfully receives a packet with a given packet number, it MUST discard all packets with higher packet numbers if they cannot be successfully unprotected with either the same key, or - if there is a key update - the next packet protection key (see Section 6.2). Similarly, a packet that appears to trigger a key update, but cannot be unprotected successfully MUST be discarded.

Failure to unprotect a packet does not necessarily indicate the existence of a protocol error in a peer or an attack. The truncated

packet number encoding used in QUIC can cause packet numbers to be decoded incorrectly if they are delayed significantly.

6. Key Phases

As TLS reports the availability of 0-RTT and 1-RTT keys, new keying material can be exported from TLS and used for QUIC packet protection. At each transition during the handshake a new secret is exported from TLS and packet protection keys are derived from that secret.

Every time that a new set of keys is used for protecting outbound packets, the KEY_PHASE bit in the public flags is toggled. The exception is the transition from 0-RTT keys to 1-RTT keys, where the presence of the version field and its associated bit is used (see Section 6.1.1).

Once the connection is fully enabled, the KEY_PHASE bit allows a recipient to detect a change in keying material without necessarily needing to receive the first packet that triggered the change. An endpoint that notices a changed KEY_PHASE bit can update keys and decrypt the packet that contains the changed bit, see Section 6.2.

The KEY_PHASE bit is the third bit of the public flags (0x04).

Transitions between keys during the handshake are complicated by the need to ensure that TLS handshake messages are sent with the correct packet protection.

6.1. Packet Protection for the TLS Handshake

The initial exchange of packets are sent without protection. These packets are marked with a KEY_PHASE of 0.

TLS handshake messages MUST NOT be protected using QUIC packet protection. A KEY_PHASE of 0 is used for all of these packets, even during retransmission. The messages affected are all TLS handshake message up to the TLS Finished that is sent by each endpoint.

Any TLS handshake messages that are sent after completing the TLS handshake do not need special packet protection rules. Packets containing these messages use the packet protection keys that are current at the time of sending (or retransmission).

Like the client, a server MUST send retransmissions of its unprotected handshake messages or acknowledgments for unprotected handshake messages sent by the client in unprotected packets (KEY_PHASE=0).

6.1.1. Initial Key Transitions

Once the TLS handshake is complete, keying material is exported from TLS and QUIC packet protection commences.

Packets protected with 1-RTT keys have a KEY_PHASE bit set to 1. These packets also have a VERSION bit set to 0.

If the client sends 0-RTT data, it marks packets protected with 0-RTT keys with a KEY_PHASE of 1 and a VERSION bit of 1. Setting the version bit means that all packets also include the version field. The client retains the VERSION bit, but reverts the KEY_PHASE bit for the packet that contains the TLS EndOfEarlyData and Finished messages.

The client clears the VERSION bit and sets the KEY_PHASE bit to 1 when it transitions to using 1-RTT keys.

Marking 0-RTT data with the both KEY_PHASE and VERSION bits ensures that the server is able to identify these packets as 0-RTT data in case packets containing TLS handshake message are lost or delayed. Including the version also ensures that the packet format is known to the server in this case.

Using both KEY_PHASE and VERSION also ensures that the server is able to distinguish between cleartext handshake packets (KEY_PHASE=0, VERSION=1), 0-RTT protected packets (KEY_PHASE=1, VERSION=1), and 1-RTT protected packets (KEY_PHASE=1, VERSION=0). Packets with all of these markings can arrive concurrently, and being able to identify each cleanly ensures that the correct packet protection keys can be selected and applied.

A server might choose to retain 0-RTT packets that arrive before a TLS ClientHello. The server can then use those packets once the ClientHello arrives. However, the potential for denial of service from buffering 0-RTT packets is significant. These packets cannot be authenticated and so might be employed by an attacker to exhaust server resources. Limiting the number of packets that are saved might be necessary.

The server transitions to using 1-RTT keys after sending its first flight of TLS handshake messages. From this point, the server protects all packets with 1-RTT keys. Future packets are therefore protected with 1-RTT keys and marked with a KEY_PHASE of 1.

6.1.2. Retransmission and Acknowledgment of Unprotected Packets

TLS handshake messages from both client and server are critical to the key exchange. The contents of these messages determines the keys used to protect later messages. If these handshake messages are included in packets that are protected with these keys, they will be indecipherable to the recipient.

Even though newer keys could be available when retranmitting, retransmissions of these handshake messages MUST be sent in unprotected packets (with a KEY_PHASE of 0). An endpoint MUST also generate ACK frames for these messages that are sent in unprotected packets.

A HelloRetryRequest handshake message might be used to reject an initial ClientHello. A HelloRetryRequest handshake message and any second ClientHello that is sent in response MUST also be sent without packet protection. This is natural, because no new keying material will be available when these messages need to be sent. Upon receipt of a HelloRetryRequest, a client SHOULD cease any transmission of 0-RTT data; 0-RTT data will only be discarded by any server that sends a HelloRetryRequest.

The KEY_PHASE and VERSION bits ensure that protected packets are clearly distinguished from unprotected packets. Loss or reordering might cause unprotected packets to arrive once 1-RTT keys are in use, unprotected packets are easily distinguished from 1-RTT packets.

Once 1-RTT keys are available to an endpoint, it no longer needs the TLS handshake messages that are carried in unprotected packets. However, a server might need to retransmit its TLS handshake messages in response to receiving an unprotected packet that contains ACK frames. A server MUST process ACK frames in unprotected packets until the TLS handshake is reported as complete, or it receives an ACK frame in a protected packet that acknowledges all of its handshake messages.

To limit the number of key phases that could be active, an endpoint MUST NOT initiate a key update while there are any unacknowledged handshake messages, see Section 6.2.

6.2. Key Update

Once the TLS handshake is complete, the KEY_PHASE bit allows for refreshes of keying material by either peer. Endpoints start using updated keys immediately without additional signaling; the change in the KEY_PHASE bit indicates that a new key is in use.

An endpoint MUST NOT initiate more than one key update at a time. A new key cannot be used until the endpoint has received and successfully decrypted a packet with a matching KEY_PHASE. Note that when 0-RTT is attempted the value of the KEY_PHASE bit will be different on packets sent by either peer.

A receiving endpoint detects an update when the KEY_PHASE bit doesn't match what it is expecting. It creates a new secret (see Section 5.2) and the corresponding read key and IV. If the packet can be decrypted and authenticated using these values, then the keys it uses for packet protection are also updated. The next packet sent by the endpoint will then use the new keys.

An endpoint doesn't need to send packets immediately when it detects that its peer has updated keys. The next packet that it sends will simply use the new keys. If an endpoint detects a second update before it has sent any packets with updated keys it indicates that its peer has updated keys twice without awaiting a reciprocal update. An endpoint MUST treat consecutive key updates as a fatal error and abort the connection.

An endpoint SHOULD retain old keys for a short period to allow it to decrypt packets with smaller packet numbers than the packet that triggered the key update. This allows an endpoint to consume packets that are reordered around the transition between keys. Packets with higher packet numbers always use the updated keys and MUST NOT be decrypted with old keys.

Keys and their corresponding secrets SHOULD be discarded when an endpoint has received all packets with sequence numbers lower than the lowest sequence number used for the new key. An endpoint might discard keys if it determines that the length of the delay to affected packets is excessive.

This ensures that once the handshake is complete, packets with the same KEY_PHASE will have the same packet protection keys, unless there are multiple key updates in a short time frame succession and significant packet reordering.

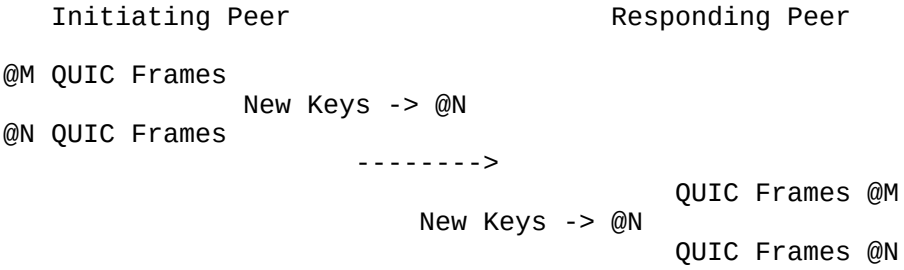


Figure 5: Key Update

As shown in Figure 3 and Figure 5, there is never a situation where there are more than two different sets of keying material that might be received by a peer. Once both sending and receiving keys have been updated,

A server cannot initiate a key update until it has received the client's Finished message. Otherwise, packets protected by the updated keys could be confused for retransmissions of handshake messages. A client cannot initiate a key update until all of its handshake messages have been acknowledged by the server.

A packet that triggers a key update could arrive after successfully processing a packet with a higher packet number. This is only possible if there is a key compromise and an attack, or if the peer is incorrectly reverting to use of old keys. Because the latter cannot be differentiated from an attack, an endpoint MUST immediately terminate the connection if it detects this condition.

7. Client Address Validation

Two tools are provided by TLS to enable validation of client source addresses at a server: the cookie in the HelloRetryRequest message, and the ticket in the NewSessionTicket message.

7.1. HelloRetryRequest Address Validation

The cookie extension in the TLS HelloRetryRequest message allows a server to perform source address validation during the handshake.

When QUIC requests address validation during the processing of the first ClientHello, the token it provides is included in the cookie extension of a HelloRetryRequest. As long as the cookie cannot be successfully guessed by a client, the server can be assured that the client received the HelloRetryRequest if it includes the value in a second ClientHello.

An initial ClientHello never includes a cookie extension. Thus, if a server constructs a cookie that contains all the information necessary to reconstruct state, it can discard local state after sending a HelloRetryRequest. Presence of a valid cookie in a ClientHello indicates that the ClientHello is a second attempt from the client.

An address validation token can be extracted from a second ClientHello and passed to the transport for further validation. If that validation fails, the server MUST fail the TLS handshake and send an illegal_parameter alert.

Combining address validation with the other uses of HelloRetryRequest ensures that there are fewer ways in which an additional round-trip can be added to the handshake. In particular, this makes it possible to combine a request for address validation with a request for a different client key share.

If TLS needs to send a HelloRetryRequest for other reasons, it needs to ensure that it can correctly identify the reason that the HelloRetryRequest was generated. During the processing of a second ClientHello, TLS does not need to consult the transport protocol regarding address validation if address validation was not requested originally. In such cases, the cookie extension could either be absent or it could indicate that an address validation token is not present.

7.2. NewSessionTicket Address Validation

The ticket in the TLS NewSessionTicket message allows a server to provide a client with a similar sort of token. When a client resumes a TLS connection - whether or not 0-RTT is attempted - it includes the ticket in the handshake message. As with the HelloRetryRequest cookie, the server includes the address validation token in the ticket. TLS provides the token it extracts from the session ticket to the transport when it asks whether source address validation is needed.

If both a HelloRetryRequest cookie and a session ticket are present in the ClientHello, only the token from the cookie is passed to the transport. The presence of a cookie indicates that this is a second ClientHello - the token from the session ticket will have been provided to the transport when it appeared in the first ClientHello.

A server can send a NewSessionTicket message at any time. This allows it to update the state - and the address validation token - that is included in the ticket. This might be done to refresh the ticket or token, or it might be generated in response to changes in

the state of the connection. QUIC can request that a NewSessionTicket be sent by providing a new address validation token.

A server that intends to support 0-RTT SHOULD provide an address validation token immediately after completing the TLS handshake.

7.3. Address Validation Token Integrity

TLS MUST provide integrity protection for address validation token unless the transport guarantees integrity protection by other means. For a NewSessionTicket that includes confidential information - such as the resumption secret - including the token under authenticated encryption ensures that the token gains both confidentiality and integrity protection without duplicating the overheads of that protection.

8. Pre-handshake QUIC Messages

Implementations MUST NOT exchange data on any stream other than stream 1 without packet protection. QUIC requires the use of several types of frame for managing loss detection and recovery during this phase. In addition, it might be useful to use the data acquired during the exchange of unauthenticated messages for congestion control.

This section generally only applies to TLS handshake messages from both peers and acknowledgments of the packets carrying those messages. In many cases, the need for servers to provide acknowledgments is minimal, since the messages that clients send are small and implicitly acknowledged by the server’s responses.

The actions that a peer takes as a result of receiving an unauthenticated packet needs to be limited. In particular, state established by these packets cannot be retained once record protection commences.

There are several approaches possible for dealing with unauthenticated packets prior to handshake completion:

- discard and ignore them
- use them, but reset any state that is established once the handshake completes
- use them and authenticate them afterwards; failing the handshake if they can’t be authenticated
- save them and use them when they can be properly authenticated

- treat them as a fatal error

Different strategies are appropriate for different types of data. This document proposes that all strategies are possible depending on the type of message.

- Transport parameters are made usable and authenticated as part of the TLS handshake (see Section 9.2).

- Most unprotected messages are treated as fatal errors when received except for the small number necessary to permit the handshake to complete (see Section 8.1).

- Protected packets can either be discarded or saved and later used (see Section 8.3).

8.1. Unprotected Packets Prior to Handshake Completion

This section describes the handling of messages that are sent and received prior to the completion of the TLS handshake.

Sending and receiving unprotected messages is hazardous. Unless expressly permitted, receipt of an unprotected message of any kind MUST be treated as a fatal error.

8.1.1. STREAM Frames

"STREAM" frames for stream 1 are permitted. These carry the TLS handshake messages. Once 1-RTT keys are available, unprotected "STREAM" frames on stream 1 can be ignored.

Receiving unprotected "STREAM" frames for other streams MUST be treated as a fatal error.

8.1.2. ACK Frames

"ACK" frames are permitted prior to the handshake being complete. Information learned from "ACK" frames cannot be entirely relied upon, since an attacker is able to inject these packets. Timing and packet retransmission information from "ACK" frames is critical to the functioning of the protocol, but these frames might be spoofed or altered.

Endpoints MUST NOT use an unprotected "ACK" frame to acknowledge data that was protected by 0-RTT or 1-RTT keys. An endpoint MUST ignore an unprotected "ACK" frame if it claims to acknowledge data that was sent in a protected packet. Such an acknowledgement can only serve

as a denial of service, since an endpoint that can read protected data is always able to send protected data.

ISSUE: What about 0-RTT data? Should we allow acknowledgment of 0-RTT with unprotected frames? If we don't, then 0-RTT data will be unacknowledged until the handshake completes. This isn't a problem if the handshake completes without loss, but it could mean that 0-RTT stalls when a handshake packet disappears for any reason.

An endpoint SHOULD use data from unprotected or 0-RTT-protected "ACK" frames only during the initial handshake and while they have insufficient information from 1-RTT-protected "ACK" frames. Once sufficient information has been obtained from protected messages, information obtained from less reliable sources can be discarded.

8.1.3. WINDOW_UPDATE Frames

"WINDOW_UPDATE" frames MUST NOT be sent unprotected.

Though data is exchanged on stream 1, the initial flow control window is sufficiently large to allow the TLS handshake to complete. This limits the maximum size of the TLS handshake and would prevent a server or client from using an abnormally large certificate chain.

Stream 1 is exempt from the connection-level flow control window.

8.1.4. Denial of Service with Unprotected Packets

Accepting unprotected - specifically unauthenticated - packets presents a denial of service risk to endpoints. An attacker that is able to inject unprotected packets can cause a recipient to drop even protected packets with a matching sequence number. The spurious packet shadows the genuine packet, causing the genuine packet to be ignored as redundant.

Once the TLS handshake is complete, both peers MUST ignore unprotected packets. From that point onward, unprotected messages can be safely dropped.

Since only TLS handshake packets and acknowledgments are sent in the clear, an attacker is able to force implementations to rely on retransmission for packets that are lost or shadowed. Thus, an attacker that intends to deny service to an endpoint has to drop or shadow protected packets in order to ensure that their victim continues to accept unprotected packets. The ability to shadow packets means that an attacker does not need to be on path.

ISSUE: This would not be an issue if QUIC had a randomized starting sequence number. If we choose to randomize, we fix this problem and reduce the denial of service exposure to on-path attackers. The only possible problem is in authenticating the initial value, so that peers can be sure that they haven't missed an initial message.

In addition to causing valid packets to be dropped, an attacker can generate packets with an intent of causing the recipient to expend processing resources. See Section 10.2 for a discussion of these risks.

To avoid receiving TLS packets that contain no useful data, a TLS implementation MUST reject empty TLS handshake records and any record that is not permitted by the TLS state machine. Any TLS application data or alerts that is received prior to the end of the handshake MUST be treated as a fatal error.

8.2. Use of 0-RTT Keys

If 0-RTT keys are available, the lack of replay protection means that restrictions on their use are necessary to avoid replay attacks on the protocol.

A client MUST only use 0-RTT keys to protect data that is idempotent. A client MAY wish to apply additional restrictions on what data it sends prior to the completion of the TLS handshake. A client otherwise treats 0-RTT keys as equivalent to 1-RTT keys.

A client that receives an indication that its 0-RTT data has been accepted by a server can send 0-RTT data until it receives all of the server's handshake messages. A client SHOULD stop sending 0-RTT data if it receives an indication that 0-RTT data has been rejected.

A server MUST NOT use 0-RTT keys to protect packets.

8.3. Receiving Out-of-Order Protected Frames

Due to reordering and loss, protected packets might be received by an endpoint before the final TLS handshake messages are received. A client will be unable to decrypt 1-RTT packets from the server, whereas a server will be able to decrypt 1-RTT packets from the client.

Packets protected with 1-RTT keys MAY be stored and later decrypted and used once the handshake is complete. A server MUST NOT use 1-RTT protected packets before verifying either the client Finished message or - in the case that the server has chosen to use a pre-shared key -

the pre-shared key binder (see Section 4.2.8 of [I-D.ietf-tls-tls13]). Verifying these values provides the server with an assurance that the ClientHello has not been modified.

A server could receive packets protected with 0-RTT keys prior to receiving a TLS ClientHello. The server MAY retain these packets for later decryption in anticipation of receiving a ClientHello.

Receiving and verifying the TLS Finished message is critical in ensuring the integrity of the TLS handshake. A server MUST NOT use protected packets from the client prior to verifying the client Finished message if its response depends on client authentication.

9. QUIC-Specific Additions to the TLS Handshake

QUIC uses the TLS handshake for more than just negotiation of cryptographic parameters. The TLS handshake validates protocol version selection, provides preliminary values for QUIC transport parameters, and allows a server to perform return routeability checks on clients.

9.1. Protocol and Version Negotiation

The QUIC version negotiation mechanism is used to negotiate the version of QUIC that is used prior to the completion of the handshake. However, this packet is not authenticated, enabling an active attacker to force a version downgrade.

To ensure that a QUIC version downgrade is not forced by an attacker, version information is copied into the TLS handshake, which provides integrity protection for the QUIC negotiation. This does not prevent version downgrade during the handshake, though it means that such a downgrade causes a handshake failure.

TLS uses Application Layer Protocol Negotiation (ALPN) [RFC7301] to select an application protocol. The application-layer protocol MAY restrict the QUIC versions that it can operate over. Servers MUST select an application protocol compatible with the QUIC version that the client has selected.

If the server cannot select a compatible combination of application protocol and QUIC version, it MUST abort the connection. A client MUST abort a connection if the server picks an incompatible combination of QUIC version and ALPN identifier.

9.2. QUIC Transport Parameters Extension

QUIC transport parameters are carried in a TLS extension. Different versions of QUIC might define a different format for this struct.

Including transport parameters in the TLS handshake provides integrity protection for these values.

```
enum {
   quic_transport_parameters(26), (65535)
} ExtensionType;
```

The "extension_data" field of the quic_transport_parameters extension contains a value that is defined by the version of QUIC that is in use. The quic_transport_parameters extension carries a TransportParameters when the version of QUIC defined in [QUIC-TRANSPORT] is used.

9.3. Priming 0-RTT

QUIC uses TLS without modification. Therefore, it is possible to use a pre-shared key that was obtained in a TLS connection over TCP to enable 0-RTT in QUIC. Similarly, QUIC can provide a pre-shared key that can be used to enable 0-RTT in TCP.

All the restrictions on the use of 0-RTT apply, with the exception of the ALPN label, which MUST only change to a label that is explicitly designated as being compatible. The client indicates which ALPN label it has chosen by placing that ALPN label first in the ALPN extension.

The certificate that the server uses MUST be considered valid for both connections, which will use different protocol stacks and could use different port numbers. For instance, HTTP/1.1 and HTTP/2 operate over TLS and TCP, whereas QUIC operates over UDP.

Source address validation is not completely portable between different protocol stacks. Even if the source IP address remains constant, the port number is likely to be different. Packet reflection attacks are still possible in this situation, though the set of hosts that can initiate these attacks is greatly reduced. A server might choose to avoid source address validation for such a connection, or allow an increase to the amount of data that it sends toward the client without source validation.

10. Security Considerations

There are likely to be some real clangers here eventually, but the current set of issues is well captured in the relevant sections of the main text.

Never assume that because it isn't in the security considerations section it doesn't affect security. Most of this document does.

10.1. Packet Reflection Attack Mitigation

A small ClientHello that results in a large block of handshake messages from a server can be used in packet reflection attacks to amplify the traffic generated by an attacker.

Certificate caching [RFC7924] can reduce the size of the server's handshake messages significantly.

QUIC requires that the packet containing a ClientHello be padded to the size of the maximum transmission unit (MTU). A server is less likely to generate a packet reflection attack if the data it sends is a small multiple of this size. A server SHOULD use a HelloRetryRequest if the size of the handshake messages it sends is likely to significantly exceed the size of the packet containing the ClientHello.

10.2. Peer Denial of Service

QUIC, TLS and HTTP/2 all contain a messages that have legitimate uses in some contexts, but that can be abused to cause a peer to expend processing resources without having any observable impact on the state of the connection. If processing is disproportionately large in comparison to the observable effects on bandwidth or state, then this could allow a malicious peer to exhaust processing capacity without consequence.

QUIC prohibits the sending of empty "STREAM" frames unless they are marked with the FIN bit. This prevents "STREAM" frames from being sent that only waste effort.

TLS records SHOULD always contain at least one octet of a handshake messages or alert. Records containing only padding are permitted during the handshake, but an excessive number might be used to generate unnecessary work. Once the TLS handshake is complete, endpoints SHOULD NOT send TLS application data records unless it is to hide the length of QUIC records. QUIC packet protection does not include any allowance for padding; padded TLS application data records can be used to mask the length of QUIC frames.

While there are legitimate uses for some redundant packets, implementations SHOULD track redundant packets and treat excessive volumes of any non-productive packets as indicative of an attack.

11. Error codes

The portion of the QUIC error code space allocated for the crypto handshake is 0xC0000000-0xFFFFFFFF. The following error codes are defined when TLS is used for the crypto handshake:

TLS_HANDSHAKE_FAILED (0xC000001C): The TLS handshake failed.

TLS_FATAL_ALERT_GENERATED (0xC000001D): A TLS fatal alert was sent, causing the TLS connection to end prematurely.

TLS_FATAL_ALERT_RECEIVED (0xC000001E): A TLS fatal alert was received, causing the TLS connection to end prematurely.

12. IANA Considerations

This document has no IANA actions. Yet.

13. References

13.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-19 (work in progress), March 2017.

[QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport".

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <http://www.rfc-editor.org/info/rfc5116>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <http://www.rfc-editor.org/info/rfc5869>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <http://www.rfc-editor.org/info/rfc7230>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <http://www.rfc-editor.org/info/rfc7301>.

13.2. Informative References

[AEBounds] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", March 2016, <http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>.

[QUIC-HTTP] Bishop, M., Ed., "Hypertext Transfer Protocol (HTTP) over QUIC".

[QUIC-RECOVERY] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control".

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <http://www.rfc-editor.org/info/rfc793>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <http://www.rfc-editor.org/info/rfc2818>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <http://www.rfc-editor.org/info/rfc5280>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <http://www.rfc-editor.org/info/rfc7540>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <http://www.rfc-editor.org/info/rfc7924>.

Appendix A. Contributors

Ryan Hamilton was originally an author of this specification.

Appendix B. Acknowledgments

This document has benefited from input from Dragana Damjanovic, Christian Huitema, Jana Iyengar, Adam Langley, Roberto Peon, Eric Rescorla, Ian Swett, and many others.

Appendix C. Change Log

> *RFC Editor's Note:* Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

C.1. Since draft-ietf-quic-tls-01:

- Use TLS alerts to signal TLS errors (#272, #374)
- Require ClientHello to fit in a single packet (#338)
- The second client handshake flight is now sent in the clear (#262, #337)
- The QUIC header is included as AEAD Associated Data (#226, #243, #302)
- Add interface necessary for client address validation (#275)
- Define peer authentication (#140)
- Require at least TLS 1.3 (#138)
- Define transport parameters as a TLS extension (#122)
- Define handling for protected packets before the handshake completes (#39)
- Decouple QUIC version and ALPN (#12)

C.2. Since draft-ietf-quic-tls-00:

- Changed bit used to signal key phase.
- Updated key phase markings during the handshake.

- Added TLS interface requirements section.
- Moved to use of TLS exporters for key derivation.
- Moved TLS error code definitions into this document.

C.3. Since draft-thomson-quic-tls-01:

- Adopted as base for draft-ietf-quic-tls.
- Updated authors/editors list.
- Added status note.

Authors' Addresses

Martin Thomson (editor)
Mozilla

Email: martin.thomson@gmail.com

Sean Turner (editor)
sn3rd

QUIC
Internet-Draft
Intended status: Standards Track
Expires: October 19, 2018

M. Thomson, Ed.
Mozilla
S. Turner, Ed.
sn3rd

April 17, 2018

Using Transport Layer Security (TLS) to Secure QUIC
draft-ietf-quic-tls-11

Abstract

This document describes how Transport Layer Security (TLS) is used to secure QUIC.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic [1].

Working Group information can be found at https://github.com/quicwg [2]; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/-tls [3].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 19, 2018.

Copyright Notice

Table of Contents

1. Introduction

This document describes how QUIC [QUIC-TRANSPORT] is secured using Transport Layer Security (TLS) version 1.3 [TLS13]. TLS 1.3 provides critical latency improvements for connection establishment over previous versions. Absent packet loss, most new connections can be established and secured within a single round trip; on subsequent connections between the same client and server, the client can often send application data immediately, that is, using a zero round trip setup.

This document describes how the standardized TLS 1.3 acts a security component of QUIC. The same design could work for TLS 1.2, though few of the benefits QUIC provides would be realized due to the handshake latency in versions of TLS prior to 1.3.

2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses the terminology established in [QUIC-TRANSPORT].

For brevity, the acronym TLS is used to refer to TLS 1.3.

TLS terminology is used when referring to parts of TLS. Though TLS assumes a continuous stream of octets, it divides that stream into _records_. Most relevant to QUIC are the records that contain TLS _handshake messages_, which are discrete messages that are used for key agreement, authentication and parameter negotiation. Ordinarily, TLS records can also contain _application data_, though in the QUIC usage there is no use of TLS application data.

3. Protocol Overview

QUIC [QUIC-TRANSPORT] assumes responsibility for the confidentiality and integrity protection of packets. For this it uses keys derived from a TLS 1.3 connection [TLS13]; QUIC also relies on TLS 1.3 for authentication and negotiation of parameters that are critical to security and performance.

Rather than a strict layering, these two protocols are co-dependent: QUIC uses the TLS handshake; TLS uses the reliability and ordered delivery provided by QUIC streams.

This document defines how QUIC interacts with TLS. This includes a description of how TLS is used, how keying material is derived from TLS, and the application of that keying material to protect QUIC packets. Figure 1 shows the basic interactions between TLS and QUIC, with the QUIC packet protection being called out specially.

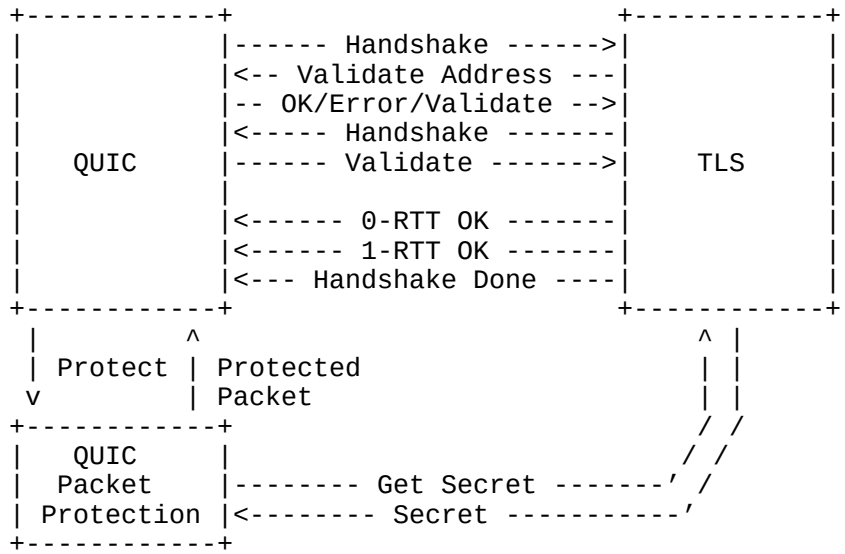


Figure 1: QUIC and TLS Interactions

The initial state of a QUIC connection has packets exchanged without any form of protection. In this state, QUIC is limited to using stream 0 and associated packets. Stream 0 is reserved for a TLS connection. This is a complete TLS connection as it would appear when layered over TCP; the only difference is that QUIC provides the reliability and ordering that would otherwise be provided by TCP.

At certain points during the TLS handshake, keying material is exported from the TLS connection for use by QUIC. This keying material is used to derive packet protection keys. Details on how and when keys are derived and used are included in Section 5.

3.1. TLS Overview

TLS provides two endpoints with a way to establish a means of communication over an untrusted medium (that is, the Internet) that ensures that messages they exchange cannot be observed, modified, or forged.

TLS features can be separated into two basic functions: an authenticated key exchange and record protection. QUIC primarily uses the authenticated key exchange provided by TLS but provides its own packet protection.

The TLS authenticated key exchange occurs between two entities: client and server. The client initiates the exchange and the server responds. If the key exchange completes successfully, both client and server will agree on a secret. TLS supports both pre-shared key (PSK) and Diffie-Hellman (DH) key exchanges. PSK is the basis for 0-RTT; the latter provides perfect forward secrecy (PFS) when the DH keys are destroyed.

After completing the TLS handshake, the client will have learned and authenticated an identity for the server and the server is optionally able to learn and authenticate an identity for the client. TLS supports X.509 [RFC5280] certificate-based authentication for both server and client.

The TLS key exchange is resistent to tampering by attackers and it produces shared secrets that cannot be controlled by either participating peer.

3.2. TLS Handshake

TLS 1.3 provides two basic handshake modes of interest to QUIC:

- A full 1-RTT handshake in which the client is able to send application data after one round trip and the server immediately responds after receiving the first handshake message from the client.

- A 0-RTT handshake in which the client uses information it has previously learned about the server to send application data immediately. This application data can be replayed by an attacker so it MUST NOT carry a self-contained trigger for any non-idempotent action.

A simplified TLS 1.3 handshake with 0-RTT application data is shown in Figure 2, see [TLS13] for more options and details.

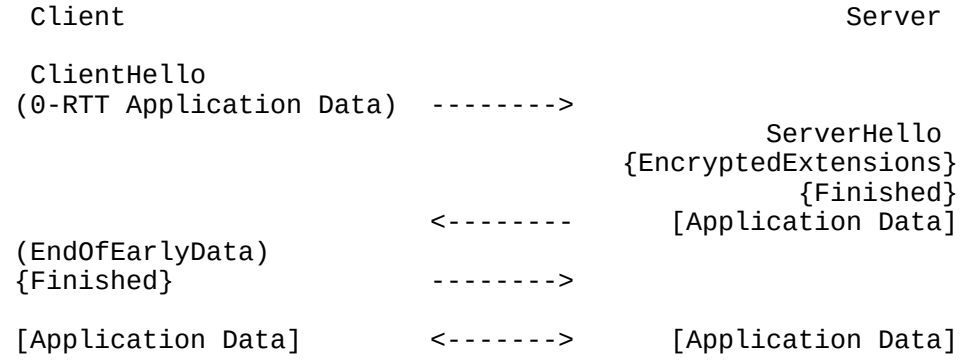


Figure 2: TLS Handshake with 0-RTT

This 0-RTT handshake is only possible if the client and server have previously communicated. In the 1-RTT handshake, the client is unable to send protected application data until it has received all of the handshake messages sent by the server.

Two additional variations on this basic handshake exchange are relevant to this document:

- The server can respond to a ClientHello with a HelloRetryRequest, which adds an additional round trip prior to the basic exchange. This is needed if the server wishes to request a different key exchange key from the client. HelloRetryRequest is also used to verify that the client is correctly able to receive packets on the address it claims to have (see [QUIC-TRANSPORT]).

- A pre-shared key mode can be used for subsequent handshakes to reduce the number of public key operations. This is the basis for 0-RTT data, even if the remainder of the connection is protected by a new Diffie-Hellman exchange.

4. TLS Usage

QUIC reserves stream 0 for a TLS connection. Stream 0 contains a complete TLS connection, which includes the TLS record layer. Other than the definition of a QUIC-specific extension (see Section 9.2), TLS is unmodified for this use. This means that TLS will apply confidentiality and integrity protection to its records. In particular, TLS record protection is what provides confidentiality protection for the TLS handshake messages sent by the server.

QUIC permits a client to send frames on streams starting from the first packet. The initial packet from a client contains a stream frame for stream 0 that contains the first TLS handshake messages

from the client. This allows the TLS handshake to start with the first packet that a client sends.

QUIC packets are protected using a scheme that is specific to QUIC, see Section 5. Keys are exported from the TLS connection when they become available using a TLS exporter (see Section 7.5 of [TLS13] and Section 5.3). After keys are exported from TLS, QUIC manages its own key schedule.

4.1. Handshake and Setup Sequence

The integration of QUIC with a TLS handshake is shown in more detail in Figure 3. QUIC "STREAM" frames on stream 0 carry the TLS handshake. QUIC performs loss recovery [QUIC-RECOVERY] for this stream and ensures that TLS handshake messages are delivered in the correct order.

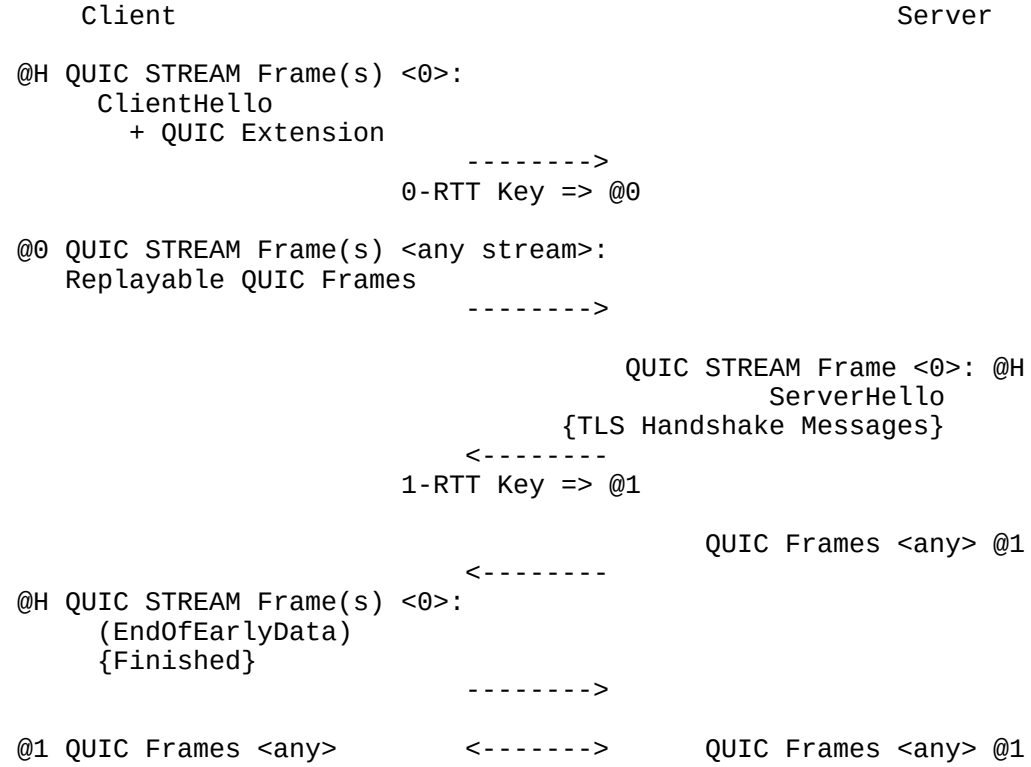


Figure 3: QUIC over TLS Handshake

In Figure 3, symbols mean:

- "<" and ">" enclose stream numbers.
- "@" indicates the keys that are used for protecting the QUIC packet (H = handshake, using keys from the well-known cleartext packet secret; 0 = 0-RTT keys; 1 = 1-RTT keys).
- "(" and ")" enclose messages that are protected with TLS 0-RTT handshake or application keys.
- "{" and "}" enclose messages that are protected by the TLS Handshake keys.

If 0-RTT is not attempted, then the client does not send packets protected by the 0-RTT key (@0). In that case, the only key transition on the client is from handshake packets (@H) to 1-RTT protection (@1), which happens after it sends its final set of TLS handshake messages.

Note: two different types of packet are used during the handshake by both client and server. The Initial packet carries a TLS ClientHello message; the remainder of the TLS handshake is carried in Handshake packets. The Retry packet carries a TLS HelloRetryRequest, if it is needed, and Handshake packets carry the remainder of the server handshake.

The server sends TLS handshake messages without protection (@H). The server transitions from no protection (@H) to full 1-RTT protection (@1) after it sends the last of its handshake messages.

Some TLS handshake messages are protected by the TLS handshake record protection. These keys are not exported from the TLS connection for use in QUIC. QUIC packets from the server are sent in the clear until the final transition to 1-RTT keys.

The client transitions from handshake (@H) to 0-RTT keys (@0) when sending 0-RTT data, and subsequently to to 1-RTT keys (@1) after its second flight of TLS handshake messages. This creates the potential for unprotected packets to be received by a server in close proximity to packets that are protected with 1-RTT keys.

More information on key transitions is included in Section 6.1.

4.2. Interface to TLS

As shown in Figure 1, the interface from QUIC to TLS consists of four primary functions: Handshake, Source Address Validation, Key Ready Events, and Secret Export.

Additional functions might be needed to configure TLS.

4.2.1. Handshake Interface

In order to drive the handshake, TLS depends on being able to send and receive handshake messages on stream 0. There are two basic functions on this interface: one where QUIC requests handshake messages and one where QUIC provides handshake packets.

Before starting the handshake QUIC provides TLS with the transport parameters (see Section 9.2) that it wishes to carry.

A QUIC client starts TLS by requesting TLS handshake octets from TLS. The client acquires handshake octets before sending its first packet.

A QUIC server starts the process by providing TLS with stream 0 octets.

Each time that an endpoint receives data on stream 0, it delivers the octets to TLS if it is able. Each time that TLS is provided with new data, new handshake octets are requested from TLS. TLS might not provide any octets if the handshake messages it has received are incomplete or it has no data to send.

At the server, when TLS provides handshake octets, it also needs to indicate whether the octets contain a HelloRetryRequest. A HelloRetryRequest MUST always be sent in a Retry packet, so the QUIC server needs to know whether the octets are a HelloRetryRequest.

Once the TLS handshake is complete, this is indicated to QUIC along with any final handshake octets that TLS needs to send. TLS also provides QUIC with the transport parameters that the peer advertised during the handshake.

Once the handshake is complete, TLS becomes passive. TLS can still receive data from its peer and respond in kind, but it will not need to send more data unless specifically requested - either by an application or QUIC. One reason to send data is that the server might wish to provide additional or updated session tickets to a client.

When the handshake is complete, QUIC only needs to provide TLS with any data that arrives on stream 0. In the same way that is done during the handshake, new data is requested from TLS after providing received data.

Important: Until the handshake is reported as complete, the connection and key exchange are not properly authenticated at the

server. Even though 1-RTT keys are available to a server after receiving the first handshake messages from a client, the server cannot consider the client to be authenticated until it receives and validates the client's Finished message.

The requirement for the server to wait for the client Finished message creates a dependency on that message being delivered. A client can avoid the potential for head-of-line blocking that this implies by sending a copy of the STREAM frame that carries the Finished message in multiple packets. This enables immediate server processing for those packets.

4.2.2. Source Address Validation

During the processing of the TLS ClientHello, TLS requests that the transport make a decision about whether to request source address validation from the client.

An initial TLS ClientHello that resumes a session includes an address validation token in the session ticket; this includes all attempts at 0-RTT. If the client does not attempt session resumption, no token will be present. While processing the initial ClientHello, TLS provides QUIC with any token that is present. In response, QUIC provides one of three responses:

- proceed with the connection,
- ask for client address validation, or
- abort the connection.

If QUIC requests source address validation, it also provides a new address validation token. TLS includes that along with any information it requires in the cookie extension of a TLS HelloRetryRequest message. In the other cases, the connection either proceeds or terminates with a handshake error.

The client echoes the cookie extension in a second ClientHello. A ClientHello that contains a valid cookie extension will always be in response to a HelloRetryRequest. If address validation was requested by QUIC, then this will include an address validation token. TLS makes a second address validation request of QUIC, including the value extracted from the cookie extension. In response to this request, QUIC cannot ask for client address validation, it can only abort or permit the connection attempt to proceed.

QUIC can provide a new address validation token for use in session resumption at any time after the handshake is complete. Each time a

new token is provided TLS generates a NewSessionTicket message, with the token included in the ticket.

See Section 7 for more details on client address validation.

4.2.3. Key Ready Events

TLS provides QUIC with signals when 0-RTT and 1-RTT keys are ready for use. These events are not asynchronous, they always occur immediately after TLS is provided with new handshake octets, or after TLS produces handshake octets.

When TLS completed its handshake, 1-RTT keys can be provided to QUIC. On both client and server, this occurs after sending the TLS Finished message.

This ordering means that there could be frames that carry TLS handshake messages ready to send at the same time that application data is available. An implementation MUST ensure that TLS handshake messages are always sent in packets protected with handshake keys (see Section 5.3.2). Separate packets are required for data that needs protection from 1-RTT keys.

If 0-RTT is possible, it is ready after the client sends a TLS ClientHello message or the server receives that message. After providing a QUIC client with the first handshake octets, the TLS stack might signal that 0-RTT keys are ready. On the server, after receiving handshake octets that contain a ClientHello message, a TLS server might signal that 0-RTT keys are available.

1-RTT keys are used for packets in both directions. 0-RTT keys are only used to protect packets sent by the client.

4.2.4. Secret Export

Details how secrets are exported from TLS are included in Section 5.3.

4.2.5. TLS Interface Summary

Figure 4 summarizes the exchange between QUIC and TLS for both client and server.

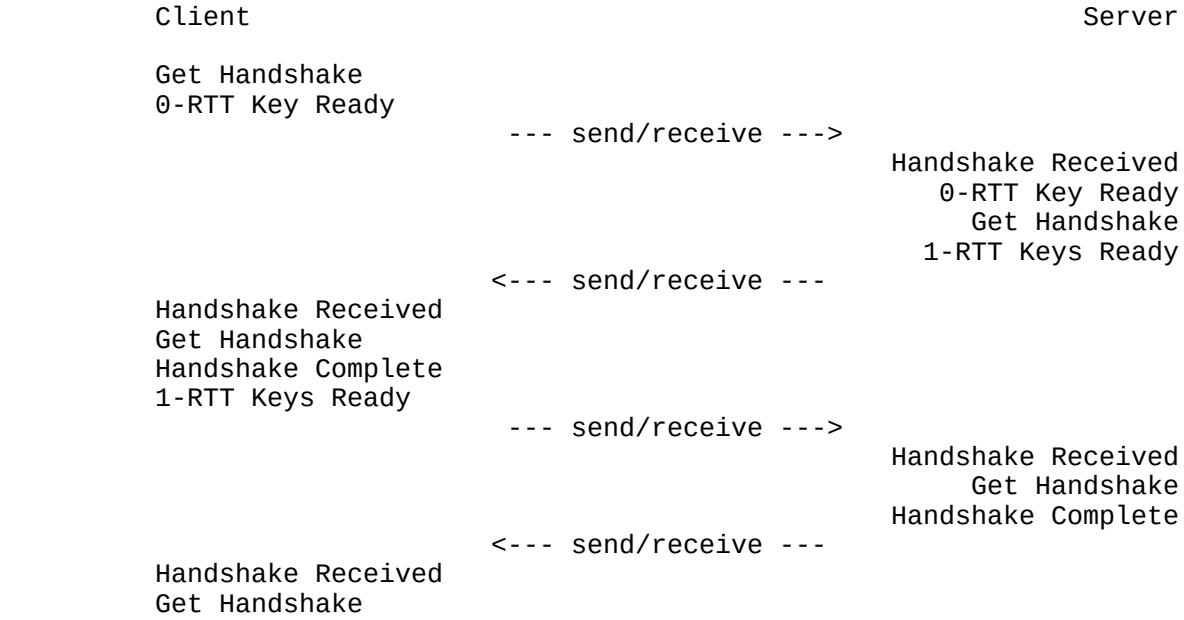


Figure 4: Interaction Summary between QUIC and TLS

4.3. TLS Version

This document describes how TLS 1.3 [TLS13] is used with QUIC.

In practice, the TLS handshake will negotiate a version of TLS to use. This could result in a newer version of TLS than 1.3 being negotiated if both endpoints support that version. This is acceptable provided that the features of TLS 1.3 that are used by QUIC are supported by the newer version.

A badly configured TLS implementation could negotiate TLS 1.2 or another older version of TLS. An endpoint MUST terminate the connection if a version of TLS older than 1.3 is negotiated.

4.4. ClientHello Size

QUIC requires that the initial handshake packet from a client fit within the payload of a single packet. The size limits on QUIC packets mean that a record containing a ClientHello needs to fit within 1129 octets, though endpoints can reduce the size of their connection ID to increase by up to 22 octets.

A TLS ClientHello can fit within this limit with ample space remaining. However, there are several variables that could cause

this limit to be exceeded. Implementations are reminded that large session tickets or HelloRetryRequest cookies, multiple or large key shares, and long lists of supported ciphers, signature algorithms, versions, QUIC transport parameters, and other negotiable parameters and extensions could cause this message to grow.

For servers, the size of the session tickets and HelloRetryRequest cookie extension can have an effect on a client's ability to connect. Choosing a small value increases the probability that these values can be successfully used by a client.

The TLS implementation does not need to ensure that the ClientHello is sufficiently large. QUIC PADDING frames are added to increase the size of the packet as necessary.

4.5. Peer Authentication

The requirements for authentication depend on the application protocol that is in use. TLS provides server authentication and permits the server to request client authentication.

A client MUST authenticate the identity of the server. This typically involves verification that the identity of the server is included in a certificate and that the certificate is issued by a trusted entity (see for example [RFC2818]).

A server MAY request that the client authenticate during the handshake. A server MAY refuse a connection if the client is unable to authenticate when requested. The requirements for client authentication vary based on application protocol and deployment.

A server MUST NOT use post-handshake client authentication (see Section 4.6.2 of [TLS13]).

4.6. Rejecting 0-RTT

A server rejects 0-RTT by rejecting 0-RTT at the TLS layer. This results in early exporter keys being unavailable, thereby preventing the use of 0-RTT for QUIC.

A client that attempts 0-RTT MUST also consider 0-RTT to be rejected if it receives a Retry or Version Negotiation packet.

When 0-RTT is rejected, all connection characteristics that the client assumed might be incorrect. This includes the choice of application protocol, transport parameters, and any application configuration. The client therefore MUST reset the state of all streams, including application state bound to those streams.

4.7. TLS Errors

Errors in the TLS connection SHOULD be signaled using TLS alerts on stream 0. A failure in the handshake MUST be treated as a QUIC connection error of type TLS_HANDSHAKE_FAILED. Once the handshake is complete, an error in the TLS connection that causes a TLS alert to be sent or received MUST be treated as a QUIC connection error of type TLS_FATAL_ALERT_GENERATED or TLS_FATAL_ALERT_RECEIVED respectively.

5. QUIC Packet Protection

QUIC packet protection provides authenticated encryption of packets. This provides confidentiality and integrity protection for the content of packets (see Section 5.4). Packet protection uses keys that are exported from the TLS connection (see Section 5.3).

Different keys are used for QUIC packet protection and TLS record protection. TLS handshake messages are protected solely with TLS record protection, but post-handshake messages are redundantly protected with both the QUIC packet protection and the TLS record protection. These messages are limited in number, and so the additional overhead is small.

5.1. Installing New Keys

As TLS reports the availability of keying material, the packet protection keys and initialization vectors (IVs) are updated (see Section 5.3). The selection of AEAD function is also updated to match the AEAD negotiated by TLS.

For packets other than any handshake packets (see Section 6.1), once a change of keys has been made, packets with higher packet numbers MUST be sent with the new keying material. The KEY_PHASE bit on these packets is inverted each time new keys are installed to signal the use of the new keys to the recipient (see Section 6 for details).

An endpoint retransmits stream data in a new packet. New packets have new packet numbers and use the latest packet protection keys. This simplifies key management when there are key updates (see Section 6.2).

5.2. Enabling 0-RTT

In order to be usable for 0-RTT, TLS MUST provide a NewSessionTicket message that contains the "max_early_data" extension with the value 0xffffffff; the amount of data which the client can send in 0-RTT is controlled by the "initial_max_data" transport parameter supplied by

the server. A client MUST treat receipt of a NewSessionTicket that contains a "max_early_data" extension with any other value as a connection error of type PROTOCOL_VIOLATION.

Early data within the TLS connection MUST NOT be used. As it is for other TLS application data, a server MUST treat receiving early data on the TLS connection as a connection error of type PROTOCOL_VIOLATION.

5.3. QUIC Key Expansion

QUIC uses a system of packet protection secrets, keys and IVs that are modelled on the system used in TLS [TLS13]. The secrets that QUIC uses as the basis of its key schedule are obtained using TLS exporters (see Section 7.5 of [TLS13]).

5.3.1. QHKDF-Expand

QUIC uses the Hash-based Key Derivation Function (HKDF) [HKDF] with the same hash function negotiated by TLS for key derivation. For example, if TLS is using the TLS_AES_128_GCM_SHA256, the SHA-256 hash function is used.

Most key derivations in this document use the QHKDF-Expand function, which uses the HKDF expand function and is modelled on the HKDF-Expand-Label function from TLS 1.3 (see Section 7.1 of [TLS13]). QHKDF-Expand differs from HKDF-Expand-Label in that it uses a different base label and omits the Context argument.

```
QHKDF-Expand(Secret, Label, Length) =
   HKDF-Expand(Secret, QhkdfExpandInfo, Length)
```

The HKDF-Expand function used by QHKDF-Expand uses the PRF hash function negotiated by TLS, except for handshake secrets and keys derived from them (see Section 5.3.2).

Where the "info" parameter of HKDF-Expand is an encoded "QhkdfExpandInfo" structure:

```
struct {
   uint16 length = Length;
   opaque label<6..255> = "QUIC " + Label;
} QhkdfExpandInfo;
```

For example, assuming a hash function with a 32 octet output, derivation for a client packet protection key would use HKDF-Expand with an "info" parameter of 0x00200855154943206b6579.

5.3.2. Handshake Secrets

Packets that carry the TLS handshake (Initial, Retry, and Handshake) are protected with a secret derived from the Destination Connection ID field from the client's Initial packet. Specifically:

```
handshake_salt = 0x9c108f98520a5c5c32968e950e8a2c5fe06d6c38
handshake_secret =
    HKDF-Extract(handshake_salt, client_dst_connection_id)

client_handshake_secret =
   QHKDF-Expand(handshake_secret, "client hs", Hash.length)
server_handshake_secret =
   QHKDF-Expand(handshake_secret, "server hs", Hash.length)
```

The hash function for HKDF when deriving handshake secrets and keys is SHA-256 [FIPS180]. The connection ID used with QHKDF-Expand is the connection ID chosen by the client.

The handshake salt is a 20 octet sequence shown in the figure in hexadecimal notation. Future versions of QUIC SHOULD generate a new salt value, thus ensuring that the keys are different for each version of QUIC. This prevents a middlebox that only recognizes one version of QUIC from seeing or modifying the contents of handshake packets from future versions.

Note: The Destination Connection ID is of arbitrary length, and it could be zero length if the server sends a Retry packet with a zero-length Source Connection ID field. In this case, the handshake keys provide no assurance to the client that the server received its packet; the client has to rely on the exchange that included the Retry packet for that property.

5.3.3. 0-RTT Secret

0-RTT keys are those keys that are used in resumed connections prior to the completion of the TLS handshake. Data sent using 0-RTT keys might be replayed and so has some restrictions on its use, see Section 8.2. 0-RTT keys are used after sending or receiving a ClientHello.

The secret is exported from TLS using the exporter label "EXPORTER-QUIC 0rtt" and an empty context. The size of the secret MUST be the size of the hash output for the PRF hash function negotiated by TLS. This uses the TLS early_exporter_secret. The QUIC 0-RTT secret is only used for protection of packets sent by the client.

```
client_0rtt_secret =
   TLS-Early-Exporter("EXPORTER-QUIC 0rtt", "", Hash.length)
```

5.3.4. 1-RTT Secrets

1-RTT keys are used by both client and server after the TLS handshake completes. There are two secrets used at any time: one is used to derive packet protection keys for packets sent by the client, the other for packet protection keys on packets sent by the server.

The initial client packet protection secret is exported from TLS using the exporter label "EXPORTER-QUIC client 1rtt"; the initial server packet protection secret uses the exporter label "EXPORTER-QUIC server 1rtt". Both exporters use an empty context. The size of the secret MUST be the size of the hash output for the PRF hash function negotiated by TLS.

```
client_pp_secret<0> =
   TLS-Exporter("EXPORTER-QUIC client 1rtt", "", Hash.length)
server_pp_secret<0> =
   TLS-Exporter("EXPORTER-QUIC server 1rtt", "", Hash.length)
```

These secrets are used to derive the initial client and server packet protection keys.

5.3.5. Updating 1-RTT Secrets

After a key update (see Section 6.2), the 1-RTT secrets are updated using QHKDF-Expand. Updated secrets are derived from the existing packet protection secret. A Label parameter of "client 1rtt" is used for the client secret and "server 1rtt" for the server. The Length is the same as the native output of the PRF hash function.

```
client_pp_secret<N+1> =
  QHKDF-Expand(client_pp_secret<N>, "client 1rtt", Hash.length)
server_pp_secret<N+1> =
  QHKDF-Expand(server_pp_secret<N>, "server 1rtt", Hash.length)
```

This allows for a succession of new secrets to be created as needed.

5.3.6. Packet Protection Keys

The complete key expansion uses a similar process for key expansion to that defined in Section 7.3 of [TLS13], using QHKDF-Expand in place of HKDF-Expand-Label. QUIC uses the AEAD function negotiated by TLS.

The packet protection key and IV used to protect the 0-RTT packets sent by a client are derived from the QUIC 0-RTT secret. The packet protection keys and IVs for 1-RTT packets sent by the client and server are derived from the current generation of client and server 1-RTT secrets (client_pp_secret<i> and server_pp_secret<i>) respectively.

The length of the QHKDF-Expand output is determined by the requirements of the AEAD function selected by TLS. The key length is the AEAD key size. As defined in Section 5.3 of [TLS13], the IV length is the larger of 8 or N_MIN (see Section 4 of [AEAD]; all ciphersuites defined in [TLS13] have N_MIN set to 12).

For any secret S, the AEAD key uses a label of "key", and the IV uses a label of "iv":

```
key = QHKDF-Expand(S, "key", key_length)
iv  = QHKDF-Expand(S, "iv", iv_length)
```

Separate keys are derived for packet protection by clients and servers. Each endpoint uses the packet protection key of its peer to remove packet protection. For example, client packet protection keys and IVs - which are also used by the server to remove the protection added by a client - for AEAD_AES_128_GCM are derived from 1-RTT secrets as follows:

```
client_pp_key<i> = QHKDF-Expand(client_pp_secret<i>, "key", 16)
client_pp_iv<i>  = QHKDF-Expand(client_pp_secret<i>, "iv", 12)
```

The QUIC record protection initially starts with keying material derived from handshake keys. For a client, when the TLS state machine reports that the ClientHello has been sent, 0-RTT keys can be generated and installed for writing, if 0-RTT is available. Finally, the TLS state machine reports completion of the handshake and 1-RTT keys can be generated and installed for writing.

5.4. QUIC AEAD Usage

The Authentication Encryption with Associated Data (AEAD) [AEAD] function used for QUIC packet protection is AEAD that is negotiated for use with the TLS connection. For example, if TLS is using the TLS_AES_128_GCM_SHA256, the AEAD_AES_128_GCM function is used.

All QUIC packets other than Version Negotiation and Stateless Reset packets are protected with an AEAD algorithm [AEAD]. Prior to establishing a shared secret, packets are protected with AEAD_AES_128_GCM and a key derived from the client’s connection ID (see Section 5.3.2). This provides protection against off-path

attackers and robustness against QUIC version unaware middleboxes, but not against on-path attackers.

All ciphersuites currently defined for TLS 1.3 - and therefore QUIC - have a 16-byte authentication tag and produce an output 16 bytes larger than their input.

Once TLS has provided a key, the contents of regular QUIC packets immediately after any TLS messages have been sent are protected by the AEAD selected by TLS.

The key, K, is either the client packet protection key (client_pp_key<i>) or the server packet protection key (server_pp_key<i>), derived as defined in Section 5.3.

The nonce, N, is formed by combining the packet protection IV (either client_pp_iv<i> or server_pp_iv<i>) with the packet number. The 64 bits of the reconstructed QUIC packet number in network byte order is left-padded with zeros to the size of the IV. The exclusive OR of the padded packet number and the IV forms the AEAD nonce.

The associated data, A, for the AEAD is the contents of the QUIC header, starting from the flags octet in either the short or long header.

The input plaintext, P, for the AEAD is the content of the QUIC frame following the header, as described in [QUIC-TRANSPORT].

The output ciphertext, C, of the AEAD is transmitted in place of P.

5.5. Packet Numbers

QUIC has a single, contiguous packet number space. In comparison, TLS restarts its sequence number each time that record protection keys are changed. The sequence number restart in TLS ensures that a compromise of the current traffic keys does not allow an attacker to truncate the data that is sent after a key update by sending additional packets under the old key (causing new packets to be discarded).

QUIC does not assume a reliable transport and is required to handle attacks where packets are dropped in other ways. QUIC is therefore not affected by this form of truncation.

The QUIC packet number is not reset and it is not permitted to go higher than its maximum value of 2^62-1. This establishes a hard limit on the number of packets that can be sent.

Some AEAD functions have limits for how many packets can be encrypted under the same key and IV (see for example [AEBounds]). This might be lower than the packet number limit. An endpoint MUST initiate a key update (Section 6.2) prior to exceeding any limit set for the AEAD that is in use.

TLS maintains a separate sequence number that is used for record protection on the connection that is hosted on stream 0. This sequence number is not visible to QUIC.

5.6. Receiving Protected Packets

Once an endpoint successfully receives a packet with a given packet number, it MUST discard all packets with higher packet numbers if they cannot be successfully unprotected with either the same key, or - if there is a key update - the next packet protection key (see Section 6.2). Similarly, a packet that appears to trigger a key update, but cannot be unprotected successfully MUST be discarded.

Failure to unprotect a packet does not necessarily indicate the existence of a protocol error in a peer or an attack. The truncated packet number encoding used in QUIC can cause packet numbers to be decoded incorrectly if they are delayed significantly.

5.7. Packet Number Gaps

Section 6.8.5.1 of [QUIC-TRANSPORT] also requires a secret to compute packet number gaps on connection ID transitions. That secret is computed as:

```
packet_number_secret =
  TLS-Exporter("EXPORTER-QUIC packet number", "", Hash.length)
```

6. Key Phases

As TLS reports the availability of 0-RTT and 1-RTT keys, new keying material can be exported from TLS and used for QUIC packet protection. At each transition during the handshake a new secret is exported from TLS and packet protection keys are derived from that secret.

Every time that a new set of keys is used for protecting outbound packets, the KEY_PHASE bit in the public flags is toggled. 0-RTT protected packets use the QUIC long header, they do not use the KEY_PHASE bit to select the correct keys (see Section 6.1.1).

Once the connection is fully enabled, the KEY_PHASE bit allows a recipient to detect a change in keying material without necessarily

needing to receive the first packet that triggered the change. An endpoint that notices a changed KEY_PHASE bit can update keys and decrypt the packet that contains the changed bit, see Section 6.2.

The KEY_PHASE bit is included as the 0x20 bit of the QUIC short header.

Transitions between keys during the handshake are complicated by the need to ensure that TLS handshake messages are sent with the correct packet protection.

6.1. Packet Protection for the TLS Handshake

The initial exchange of packets that carry the TLS handshake are AEAD-protected using the handshake secrets generated as described in Section 5.3.2. All TLS handshake messages up to the TLS Finished message sent by either endpoint use packets protected with handshake keys.

Any TLS handshake messages that are sent after completing the TLS handshake do not need special packet protection rules. Packets containing these messages use the packet protection keys that are current at the time of sending (or retransmission).

Like the client, a server MUST send retransmissions of its unprotected handshake messages or acknowledgments for unprotected handshake messages sent by the client in packets protected with handshake keys.

6.1.1. Initial Key Transitions

Once the TLS handshake is complete, keying material is exported from TLS and used to protect QUIC packets.

Packets protected with 1-RTT keys initially have a KEY_PHASE bit set to 0. This bit inverts with each subsequent key update (see Section 6.2).

If the client sends 0-RTT data, it uses the 0-RTT packet type. The packet that contains the TLS EndOfEarlyData and Finished messages are sent in packets protected with handshake keys.

Using distinct packet types during the handshake for handshake messages, 0-RTT data, and 1-RTT data ensures that the server is able to distinguish between the different keys used to remove packet protection. All of these packets can arrive concurrently at a server.

A server might choose to retain 0-RTT packets that arrive before a TLS ClientHello. The server can then use those packets once the ClientHello arrives. However, the potential for denial of service from buffering 0-RTT packets is significant. These packets cannot be authenticated and so might be employed by an attacker to exhaust server resources. Limiting the number of packets that are saved might be necessary.

The server transitions to using 1-RTT keys after sending its first flight of TLS handshake messages, ending in the Finished. From this point, the server protects all packets with 1-RTT keys. Future packets are therefore protected with 1-RTT keys. Initially, these are marked with a KEY_PHASE of 0.

6.1.2. Retransmission and Acknowledgment of Unprotected Packets

TLS handshake messages from both client and server are critical to the key exchange. The contents of these messages determine the keys used to protect later messages. If these handshake messages are included in packets that are protected with these keys, they will be indecipherable to the recipient.

Even though newer keys could be available when retransmitting, retransmissions of these handshake messages MUST be sent in packets protected with handshake keys. An endpoint MUST generate ACK frames for these messages and send them in packets protected with handshake keys.

A HelloRetryRequest handshake message might be used to reject an initial ClientHello. A HelloRetryRequest handshake message is sent in a Retry packet; any second ClientHello that is sent in response uses a Initial packet type. These packets are only protected with a predictable key (see Section 5.3.2). This is natural, because no shared secret will be available when these messages need to be sent. Upon receipt of a HelloRetryRequest, a client SHOULD cease any transmission of 0-RTT data; 0-RTT data will only be discarded by any server that sends a HelloRetryRequest.

The packet type ensures that protected packets are clearly distinguished from unprotected packets. Loss or reordering might cause unprotected packets to arrive once 1-RTT keys are in use, unprotected packets are easily distinguished from 1-RTT packets using the packet type.

Once 1-RTT keys are available to an endpoint, it no longer needs the TLS handshake messages that are carried in unprotected packets. However, a server might need to retransmit its TLS handshake messages in response to receiving an unprotected packet that contains ACK

frames. A server MUST process ACK frames in unprotected packets until the TLS handshake is reported as complete, or it receives an ACK frame in a protected packet that acknowledges all of its handshake messages.

To limit the number of key phases that could be active, an endpoint MUST NOT initiate a key update while there are any unacknowledged handshake messages, see Section 6.2.

6.2. Key Update

Once the TLS handshake is complete, the KEY_PHASE bit allows for refreshes of keying material by either peer. Endpoints start using updated keys immediately without additional signaling; the change in the KEY_PHASE bit indicates that a new key is in use.

An endpoint MUST NOT initiate more than one key update at a time. A new key cannot be used until the endpoint has received and successfully decrypted a packet with a matching KEY_PHASE. Note that when 0-RTT is attempted the value of the KEY_PHASE bit will be different on packets sent by either peer.

A receiving endpoint detects an update when the KEY_PHASE bit doesn't match what it is expecting. It creates a new secret (see Section 5.3) and the corresponding read key and IV. If the packet can be decrypted and authenticated using these values, then the keys it uses for packet protection are also updated. The next packet sent by the endpoint will then use the new keys.

An endpoint doesn't need to send packets immediately when it detects that its peer has updated keys. The next packet that it sends will simply use the new keys. If an endpoint detects a second update before it has sent any packets with updated keys it indicates that its peer has updated keys twice without awaiting a reciprocal update. An endpoint MUST treat consecutive key updates as a fatal error and abort the connection.

An endpoint SHOULD retain old keys for a short period to allow it to decrypt packets with smaller packet numbers than the packet that triggered the key update. This allows an endpoint to consume packets that are reordered around the transition between keys. Packets with higher packet numbers always use the updated keys and MUST NOT be decrypted with old keys.

Keys and their corresponding secrets SHOULD be discarded when an endpoint has received all packets with packet numbers lower than the lowest packet number used for the new key. An endpoint might discard

keys if it determines that the length of the delay to affected packets is excessive.

This ensures that once the handshake is complete, packets with the same KEY_PHASE will have the same packet protection keys, unless there are multiple key updates in a short time frame succession and significant packet reordering.

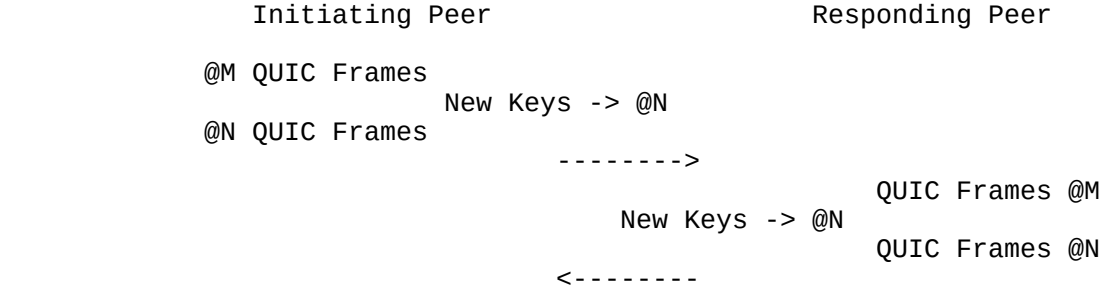


Figure 5: Key Update

As shown in Figure 3 and Figure 5, there is never a situation where there are more than two different sets of keying material that might be received by a peer. Once both sending and receiving keys have been updated, the peers immediately begin to use them.

A server cannot initiate a key update until it has received the client's Finished message. Otherwise, packets protected by the updated keys could be confused for retransmissions of handshake messages. A client cannot initiate a key update until all of its handshake messages have been acknowledged by the server.

A packet that triggers a key update could arrive after successfully processing a packet with a higher packet number. This is only possible if there is a key compromise and an attack, or if the peer is incorrectly reverting to use of old keys. Because the latter cannot be differentiated from an attack, an endpoint MUST immediately terminate the connection if it detects this condition.

7. Client Address Validation

Two tools are provided by TLS to enable validation of client source addresses at a server: the cookie in the HelloRetryRequest message, and the ticket in the NewSessionTicket message.

7.1. HelloRetryRequest Address Validation

The cookie extension in the TLS HelloRetryRequest message allows a server to perform source address validation during the handshake.

When QUIC requests address validation during the processing of the first ClientHello, the token it provides is included in the cookie extension of a HelloRetryRequest. As long as the cookie cannot be successfully guessed by a client, the server can be assured that the client received the HelloRetryRequest if it includes the value in a second ClientHello.

An initial ClientHello never includes a cookie extension. Thus, if a server constructs a cookie that contains all the information necessary to reconstruct state, it can discard local state after sending a HelloRetryRequest. Presence of a valid cookie in a ClientHello indicates that the ClientHello is a second attempt from the client.

An address validation token can be extracted from a second ClientHello and passed to the transport for further validation. If that validation fails, the server MUST fail the TLS handshake and send an illegal_parameter alert.

Combining address validation with the other uses of HelloRetryRequest ensures that there are fewer ways in which an additional round-trip can be added to the handshake. In particular, this makes it possible to combine a request for address validation with a request for a different client key share.

If TLS needs to send a HelloRetryRequest for other reasons, it needs to ensure that it can correctly identify the reason that the HelloRetryRequest was generated. During the processing of a second ClientHello, TLS does not need to consult the transport protocol regarding address validation if address validation was not requested originally. In such cases, the cookie extension could either be absent or it could indicate that an address validation token is not present.

7.1.1. Stateless Address Validation

A server can use the cookie extension to store all state necessary to continue the connection. This allows a server to avoid committing state for clients that have unvalidated source addresses.

For instance, a server could use a statically-configured key to encrypt the information that it requires and include that information in the cookie. In addition to address validation information, a

server that uses encryption also needs to be able recover the hash of the ClientHello and its length, plus any information it needs in order to reconstruct the HelloRetryRequest.

7.1.2. Sending HelloRetryRequest

A server does not need to maintain state for the connection when sending a HelloRetryRequest message. This might be necessary to avoid creating a denial of service exposure for the server. However, this means that information about the transport will be lost at the server. This includes the stream offset of stream 0, the packet number that the server selects, and any opportunity to measure round trip time.

A server MUST send a TLS HelloRetryRequest in a Retry packet. Using a Retry packet causes the client to reset stream offsets. It also avoids the need for the server select an initial packet number, which would need to be remembered so that subsequent packets could be correctly numbered.

A HelloRetryRequest message MUST NOT be split between multiple Retry packets. This means that HelloRetryRequest is subject to the same size constraints as a ClientHello (see Section 4.4).

A client might send multiple Initial packets in response to loss. If a server sends a Retry packet in response to an Initial packet, it does not have to generate the same Retry packet each time. Variations in Retry packet, if used by a client, could lead to multiple connections derived from the same ClientHello. Reuse of the client nonce is not supported by TLS and could lead to security vulnerabilities. Clients that receive multiple Retry packets MUST use only one and discard the remainder.

7.2. NewSessionTicket Address Validation

The ticket in the TLS NewSessionTicket message allows a server to provide a client with a similar sort of token. When a client resumes a TLS connection - whether or not 0-RTT is attempted - it includes the ticket in the handshake message. As with the HelloRetryRequest cookie, the server includes the address validation token in the ticket. TLS provides the token it extracts from the session ticket to the transport when it asks whether source address validation is needed.

If both a HelloRetryRequest cookie and a session ticket are present in the ClientHello, only the token from the cookie is passed to the transport. The presence of a cookie indicates that this is a second

ClientHello - the token from the session ticket will have been provided to the transport when it appeared in the first ClientHello.

A server can send a NewSessionTicket message at any time. This allows it to update the state - and the address validation token - that is included in the ticket. This might be done to refresh the ticket or token, or it might be generated in response to changes in the state of the connection. QUIC can request that a NewSessionTicket be sent by providing a new address validation token.

A server that intends to support 0-RTT SHOULD provide an address validation token immediately after completing the TLS handshake.

7.3. Address Validation Token Integrity

TLS MUST provide integrity protection for address validation token unless the transport guarantees integrity protection by other means. For a NewSessionTicket that includes confidential information - such as the resumption secret - including the token under authenticated encryption ensures that the token gains both confidentiality and integrity protection without duplicating the overheads of that protection.

8. Pre-handshake QUIC Messages

Implementations MUST NOT exchange data on any stream other than stream 0 without packet protection. QUIC requires the use of several types of frame for managing loss detection and recovery during this phase. In addition, it might be useful to use the data acquired during the exchange of unauthenticated messages for congestion control.

This section generally only applies to TLS handshake messages from both peers and acknowledgments of the packets carrying those messages. In many cases, the need for servers to provide acknowledgments is minimal, since the messages that clients send are small and implicitly acknowledged by the server’s responses.

The actions that a peer takes as a result of receiving an unauthenticated packet needs to be limited. In particular, state established by these packets cannot be retained once record protection commences.

There are several approaches possible for dealing with unauthenticated packets prior to handshake completion:

- discard and ignore them

- use them, but reset any state that is established once the handshake completes
- use them and authenticate them afterwards; failing the handshake if they can't be authenticated
- save them and use them when they can be properly authenticated
- treat them as a fatal error

Different strategies are appropriate for different types of data. This document proposes that all strategies are possible depending on the type of message.

- Transport parameters are made usable and authenticated as part of the TLS handshake (see Section 9.2).
- Most unprotected messages are treated as fatal errors when received except for the small number necessary to permit the handshake to complete (see Section 8.1).
- Protected packets can either be discarded or saved and later used (see Section 8.3).

8.1. Unprotected Packets Prior to Handshake Completion

This section describes the handling of messages that are sent and received prior to the completion of the TLS handshake.

Sending and receiving unprotected messages is hazardous. Unless expressly permitted, receipt of an unprotected message of any kind MUST be treated as a fatal error.

8.1.1. STREAM Frames

"STREAM" frames for stream 0 are permitted. These carry the TLS handshake messages. Once 1-RTT keys are available, unprotected "STREAM" frames on stream 0 can be ignored.

Receiving unprotected "STREAM" frames for other streams MUST be treated as a fatal error.

8.1.2. ACK Frames

"ACK" frames are permitted prior to the handshake being complete. Information learned from "ACK" frames cannot be entirely relied upon, since an attacker is able to inject these packets. Timing and packet retransmission information from "ACK" frames is critical to the

functioning of the protocol, but these frames might be spoofed or altered.

Endpoints MUST NOT use an "ACK" frame in an unprotected packet to acknowledge packets that were protected by 0-RTT or 1-RTT keys. An endpoint MUST treat receipt of an "ACK" frame in an unprotected packet that claims to acknowledge protected packets as a connection error of type OPTIMISTIC_ACK. An endpoint that can read protected data is always able to send protected data.

Note: 0-RTT data can be acknowledged by the server as it receives it, but any packets containing acknowledgments of 0-RTT data cannot have packet protection removed by the client until the TLS handshake is complete. The 1-RTT keys necessary to remove packet protection cannot be derived until the client receives all server handshake messages.

An endpoint SHOULD use data from "ACK" frames carried in unprotected packets or packets protected with 0-RTT keys only during the initial handshake. All "ACK" frames contained in unprotected packets that are received after successful receipt of a packet protected with 1-RTT keys MUST be discarded. An endpoint SHOULD therefore include acknowledgments for unprotected and any packets protected with 0-RTT keys until it sees an acknowledgment for a packet that is both protected with 1-RTT keys and contains an "ACK" frame.

8.1.3. Updates to Data and Stream Limits

"MAX_DATA", "MAX_STREAM_DATA", "BLOCKED", "STREAM_BLOCKED", and "MAX_STREAM_ID" frames MUST NOT be sent unprotected.

Though data is exchanged on stream 0, the initial flow control window on that stream is sufficiently large to allow the TLS handshake to complete. This limits the maximum size of the TLS handshake and would prevent a server or client from using an abnormally large certificate chain.

Stream 0 is exempt from the connection-level flow control window.

Consequently, there is no need to signal being blocked on flow control.

Similarly, there is no need to increase the number of allowed streams until the handshake completes.

8.1.4. Handshake Failures

The "CONNECTION_CLOSE" frame MAY be sent by either endpoint in a Handshake packet. This allows an endpoint to signal a fatal error with connection establishment. A "STREAM" frame carrying a TLS alert MAY be included in the same packet.

8.1.5. Address Verification

In order to perform source-address verification before the handshake is complete, "PATH_CHALLENGE" and "PATH_RESPONSE" frames MAY be exchanged unprotected.

8.1.6. Denial of Service with Unprotected Packets

Accepting unprotected - specifically unauthenticated - packets presents a denial of service risk to endpoints. An attacker that is able to inject unprotected packets can cause a recipient to drop even protected packets with a matching packet number. The spurious packet shadows the genuine packet, causing the genuine packet to be ignored as redundant.

Once the TLS handshake is complete, both peers MUST ignore unprotected packets. From that point onward, unprotected messages can be safely dropped.

Since only TLS handshake packets and acknowledgments are sent in the clear, an attacker is able to force implementations to rely on retransmission for packets that are lost or shadowed. Thus, an attacker that intends to deny service to an endpoint has to drop or shadow protected packets in order to ensure that their victim continues to accept unprotected packets. The ability to shadow packets means that an attacker does not need to be on path.

In addition to causing valid packets to be dropped, an attacker can generate packets with an intent of causing the recipient to expend processing resources. See Section 10.2 for a discussion of these risks.

To avoid receiving TLS packets that contain no useful data, a TLS implementation MUST reject empty TLS handshake records and any record that is not permitted by the TLS state machine. Any TLS application data or alerts that are received prior to the end of the handshake MUST be treated as a connection error of type PROTOCOL_VIOLATION.

8.2. Use of 0-RTT Keys

If 0-RTT keys are available (see Section 5.2), the lack of replay protection means that restrictions on their use are necessary to avoid replay attacks on the protocol.

A client MUST only use 0-RTT keys to protect data that is idempotent. A client MAY wish to apply additional restrictions on what data it sends prior to the completion of the TLS handshake. A client otherwise treats 0-RTT keys as equivalent to 1-RTT keys.

A client that receives an indication that its 0-RTT data has been accepted by a server can send 0-RTT data until it receives all of the server's handshake messages. A client SHOULD stop sending 0-RTT data if it receives an indication that 0-RTT data has been rejected.

A server MUST NOT use 0-RTT keys to protect packets.

If a server rejects 0-RTT, then the TLS stream will not include any TLS records protected with 0-RTT keys.

8.3. Receiving Out-of-Order Protected Frames

Due to reordering and loss, protected packets might be received by an endpoint before the final TLS handshake messages are received. A client will be unable to decrypt 1-RTT packets from the server, whereas a server will be able to decrypt 1-RTT packets from the client.

Packets protected with 1-RTT keys MAY be stored and later decrypted and used once the handshake is complete. A server MUST NOT use 1-RTT protected packets before verifying either the client Finished message or - in the case that the server has chosen to use a pre-shared key - the pre-shared key binder (see Section 4.2.8 of [TLS13]). Verifying these values provides the server with an assurance that the ClientHello has not been modified.

A server could receive packets protected with 0-RTT keys prior to receiving a TLS ClientHello. The server MAY retain these packets for later decryption in anticipation of receiving a ClientHello.

Receiving and verifying the TLS Finished message is critical in ensuring the integrity of the TLS handshake. A server MUST NOT use protected packets from the client prior to verifying the client Finished message if its response depends on client authentication.

9. QUIC-Specific Additions to the TLS Handshake

QUIC uses the TLS handshake for more than just negotiation of cryptographic parameters. The TLS handshake validates protocol version selection, provides preliminary values for QUIC transport parameters, and allows a server to perform return routeability checks on clients.

9.1. Protocol and Version Negotiation

The QUIC version negotiation mechanism is used to negotiate the version of QUIC that is used prior to the completion of the handshake. However, this packet is not authenticated, enabling an active attacker to force a version downgrade.

To ensure that a QUIC version downgrade is not forced by an attacker, version information is copied into the TLS handshake, which provides integrity protection for the QUIC negotiation. This does not prevent version downgrade prior to the completion of the handshake, though it means that a downgrade causes a handshake failure.

TLS uses Application Layer Protocol Negotiation (ALPN) [RFC7301] to select an application protocol. The application-layer protocol MAY restrict the QUIC versions that it can operate over. Servers MUST select an application protocol compatible with the QUIC version that the client has selected.

If the server cannot select a compatible combination of application protocol and QUIC version, it MUST abort the connection. A client MUST abort a connection if the server picks an incompatible combination of QUIC version and ALPN identifier.

9.2. QUIC Transport Parameters Extension

QUIC transport parameters are carried in a TLS extension. Different versions of QUIC might define a different format for this struct.

Including transport parameters in the TLS handshake provides integrity protection for these values.

```
enum {
   quic_transport_parameters(26), (65535)
} ExtensionType;
```

The "extension_data" field of the quic_transport_parameters extension contains a value that is defined by the version of QUIC that is in use. The quic_transport_parameters extension carries a

TransportParameters when the version of QUIC defined in [QUIC-TRANSPORT] is used.

The quic_transport_parameters extension is carried in the ClientHello and the EncryptedExtensions messages during the handshake.

10. Security Considerations

There are likely to be some real clangers here eventually, but the current set of issues is well captured in the relevant sections of the main text.

Never assume that because it isn't in the security considerations section it doesn't affect security. Most of this document does.

10.1. Packet Reflection Attack Mitigation

A small ClientHello that results in a large block of handshake messages from a server can be used in packet reflection attacks to amplify the traffic generated by an attacker.

Certificate caching [RFC7924] can reduce the size of the server's handshake messages significantly.

QUIC requires that the packet containing a ClientHello be padded to a minimum size. A server is less likely to generate a packet reflection attack if the data it sends is a small multiple of this size. A server SHOULD use a HelloRetryRequest if the size of the handshake messages it sends is likely to significantly exceed the size of the packet containing the ClientHello.

10.2. Peer Denial of Service

QUIC, TLS and HTTP/2 all contain a messages that have legitimate uses in some contexts, but that can be abused to cause a peer to expend processing resources without having any observable impact on the state of the connection. If processing is disproportionately large in comparison to the observable effects on bandwidth or state, then this could allow a malicious peer to exhaust processing capacity without consequence.

QUIC prohibits the sending of empty "STREAM" frames unless they are marked with the FIN bit. This prevents "STREAM" frames from being sent that only waste effort.

TLS records SHOULD always contain at least one octet of a handshake messages or alert. Records containing only padding are permitted during the handshake, but an excessive number might be used to

generate unnecessary work. Once the TLS handshake is complete, endpoints MUST NOT send TLS application data records. Receiving TLS application data MUST be treated as a connection error of type PROTOCOL_VIOLATION.

While there are legitimate uses for some redundant packets, implementations SHOULD track redundant packets and treat excessive volumes of any non-productive packets as indicative of an attack.

11. Error Codes

This section defines error codes from the error code space used in [QUIC-TRANSPORT].

The following error codes are defined when TLS is used for the crypto handshake:

TLS_HANDSHAKE_FAILED (0x201): The TLS handshake failed.

TLS_FATAL_ALERT_GENERATED (0x202): A TLS fatal alert was sent, causing the TLS connection to end prematurely.

TLS_FATAL_ALERT_RECEIVED (0x203): A TLS fatal alert was received, causing the TLS connection to end prematurely.

12. IANA Considerations

This document does not create any new IANA registries, but it registers the values in the following registries:

- QUIC Transport Error Codes Registry [QUIC-TRANSPORT] - IANA is to register the three error codes found in Section 11, these are summarized in Table 1.

- TLS ExtensionsType Registry [TLS-REGISTRIES] - IANA is to register the quic_transport_parameters extension found in Section 9.2. Assigning 26 to the extension would be greatly appreciated. The Recommended column is to be marked Yes. The TLS 1.3 Column is to include CH and EE.

- TLS Exporter Label Registry [TLS-REGISTRIES] - IANA is requested to register "EXPORTER-QUIC 0rtt" from Section 5.3.3; "EXPORTER-QUIC client 1rtt" and "EXPORTER-QUIC server 1-RTT" from Section 5.3.4. The DTLS column is to be marked No. The Recommended column is to be marked Yes.

Value	Error	Description	Specification
0x201	TLS_HANDSHAKE_FAILED	TLS handshake failure	Section 11
0x202	TLS_FATAL_ALERT_GENERATED	Sent TLS alert	Section 11
0x203	TLS_FATAL_ALERT_RECEIVED	Receives TLS alert	Section 11

Table 1: QUIC Transport Error Codes for TLS

13. References

13.1. Normative References

[AEAD] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[FIPS180] Department of Commerce, National., "NIST FIPS 180-4, Secure Hash Standard", March 2012, <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>.

[HKDF] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-11 (work in progress), April 2018.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[TLS-REGISTRIES] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", draft-ietf-tls-iana-registry-updates-04 (work in progress), February 2018.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-21 (work in progress), July 2017.

13.2. Informative References

[AEBounds] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", March 2016, <http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>.

[QUIC-HTTP] Bishop, M., Ed., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-11 (work in progress), April 2018.

[QUIC-RECOVERY] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", draft-ietf-quic-recovery-11 (work in progress), April 2018.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <https://www.rfc-editor.org/info/rfc2818>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <https://www.rfc-editor.org/info/rfc7924>.

13.3. URIs

[1] https://mailarchive.ietf.org/arch/search/?email_list=quic

[2] https://github.com/quicwg

[3] https://github.com/quicwg/base-drafts/labels/-tls

Appendix A. Contributors

Ryan Hamilton was originally an author of this specification.

Appendix B. Acknowledgments

This document has benefited from input from Dragana Damjanovic, Christian Huitema, Jana Iyengar, Adam Langley, Roberto Peon, Eric Rescorla, Ian Swett, and many others.

Appendix C. Change Log

> *RFC Editor's Note:* Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

C.1. Since draft-ietf-quic-tls-10

- No significant changes.

C.2. Since draft-ietf-quic-tls-09

- Cleaned up key schedule and updated the salt used for handshake packet protection (#1077)

C.3. Since draft-ietf-quic-tls-08

- Specify value for max_early_data_size to enable 0-RTT (#942)
- Update key derivation function (#1003, #1004)

C.4. Since draft-ietf-quic-tls-07

- Handshake errors can be reported with CONNECTION_CLOSE (#608, #891)

C.5. Since draft-ietf-quic-tls-05

No significant changes.

C.6. Since draft-ietf-quic-tls-04

- Update labels used in HKDF-Expand-Label to match TLS 1.3 (#642)

C.7. Since draft-ietf-quic-tls-03

No significant changes.

C.8. Since draft-ietf-quic-tls-02

- Updates to match changes in transport draft

C.9. Since draft-ietf-quic-tls-01

- Use TLS alerts to signal TLS errors (#272, #374)
- Require ClientHello to fit in a single packet (#338)
- The second client handshake flight is now sent in the clear (#262, #337)
- The QUIC header is included as AEAD Associated Data (#226, #243, #302)
- Add interface necessary for client address validation (#275)
- Define peer authentication (#140)
- Require at least TLS 1.3 (#138)
- Define transport parameters as a TLS extension (#122)
- Define handling for protected packets before the handshake completes (#39)
- Decouple QUIC version and ALPN (#12)

C.10. Since draft-ietf-quic-tls-00

- Changed bit used to signal key phase
- Updated key phase markings during the handshake
- Added TLS interface requirements section

- Moved to use of TLS exporters for key derivation
- Moved TLS error code definitions into this document

C.11. Since draft-thomson-quic-tls-01

- Adopted as base for draft-ietf-quic-tls
- Updated authors/editors list
- Added status note

Authors' Addresses

Martin Thomson (editor)
Mozilla

Email: martin.thomson@gmail.com

Sean Turner (editor)
sn3rd

Email: sean@sn3rd.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

J. Iyengar, Ed.
Google
M. Thomson, Ed.
Mozilla

March 13, 2017

QUIC: A UDP-Based Multiplexed and Secure Transport
draft-ietf-quic-transport-02

Abstract

This document defines the core of the QUIC transport protocol. This document describes connection establishment, packet format, multiplexing and reliability. Accompanying documents describe the cryptographic handshake and loss detection.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic .

Working Group information can be found at https://github.com/quicwg ; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/transport .

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 14, 2017.

Table of Contents

1. Introduction

QUIC is a multiplexed and secure transport protocol that runs on top of UDP. QUIC aims to provide a flexible set of features that allow it to be a general-purpose transport for multiple applications.

QUIC implements techniques learned from experience with TCP, SCTP and other transport protocols. Using UDP as the substrate, QUIC seeks to be compatible with legacy clients and middleboxes. QUIC authenticates all of its headers and encrypts most of the data it exchanges, including its signaling. This allows the protocol to evolve without incurring a dependency on upgrades to middleboxes.

This document describes the core QUIC protocol, including the conceptual design, wire format, and mechanisms of the QUIC protocol for connection establishment, stream multiplexing, stream and connection-level flow control, and data reliability.

Accompanying documents describe QUIC's loss detection and congestion control [QUIC-RECOVERY], and the use of TLS 1.3 for key negotiation [QUIC-TLS].

2. Conventions and Definitions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

Definitions of terms that are used in this document:

Client: The endpoint initiating a QUIC connection.

Server: The endpoint accepting incoming QUIC connections.

Endpoint: The client or server end of a connection.

Stream: A logical, bi-directional channel of ordered bytes within a QUIC connection.

Connection: A conversation between two QUIC endpoints with a single encryption context that multiplexes streams within it.

Connection ID: The identifier for a QUIC connection.

QUIC packet: A well-formed UDP payload that can be parsed by a QUIC receiver. QUIC packet size in this document refers to the UDP payload size.

2.1. Notational Conventions

Packet and frame diagrams use the format described in [RFC2360] Section 3.1, with the following additional conventions:

[x] Indicates that x is optional

{x} Indicates that x is encrypted

x (A) Indicates that x is A bits long

x (A/B/C) ... Indicates that x is one of A, B, or C bits long

x (*) ... Indicates that x is variable-length

3. A QUIC Overview

This section briefly describes QUIC’s key mechanisms and benefits. Key strengths of QUIC include:

- Low-latency connection establishment
- Multiplexing without head-of-line blocking
- Authenticated and encrypted header and payload
- Rich signaling for congestion control and loss recovery
- Stream and connection flow control
- Connection migration and resilience to NAT rebinding
- Version negotiation

3.1. Low-Latency Connection Establishment

QUIC relies on a combined cryptographic and transport handshake for setting up a secure transport connection. QUIC connections are expected to commonly use 0-RTT handshakes, meaning that for most QUIC connections, data can be sent immediately following the client handshake packet, without waiting for a reply from the server. QUIC provides a dedicated stream (Stream ID 1) to be used for performing the cryptographic handshake and QUIC options negotiation. The format of the QUIC options and parameters used during negotiation are described in this document, but the handshake protocol that runs on Stream ID 1 is described in the accompanying cryptographic handshake draft [QUIC-TLS].

3.2. Stream Multiplexing

When application messages are transported over TCP, independent application messages can suffer from head-of-line blocking. When an application multiplexes many streams atop TCP's single-bytestream abstraction, a loss of a TCP segment results in blocking of all subsequent segments until a retransmission arrives, irrespective of the application streams that are encapsulated in subsequent segments. QUIC ensures that lost packets carrying data for an individual stream only impact that specific stream. Data received on other streams can continue to be reassembled and delivered to the application.

3.3. Rich Signaling for Congestion Control and Loss Recovery

QUIC's packet framing and acknowledgments carry rich information that help both congestion control and loss recovery in fundamental ways. Each QUIC packet carries a new packet number, including those carrying retransmitted data. This obviates the need for a separate mechanism to distinguish acknowledgments for retransmissions from those for original transmissions, avoiding TCP's retransmission ambiguity problem. QUIC acknowledgments also explicitly encode the delay between the receipt of a packet and its acknowledgment being sent, and together with the monotonically-increasing packet numbers, this allows for precise network roundtrip-time (RTT) calculation. QUIC's ACK frames support up to 256 ACK blocks, so QUIC is more resilient to reordering than TCP with SACK support, as well as able to keep more bytes on the wire when there is reordering or loss.

3.4. Stream and Connection Flow Control

QUIC implements stream- and connection-level flow control, closely following HTTP/2's flow control mechanisms. At a high level, a QUIC receiver advertises the absolute byte offset within each stream up to which the receiver is willing to receive data. As data is sent,

received, and delivered on a particular stream, the receiver sends WINDOW_UPDATE frames that increase the advertised offset limit for that stream, allowing the peer to send more data on that stream. In addition to this stream-level flow control, QUIC implements connection-level flow control to limit the aggregate buffer that a QUIC receiver is willing to allocate to all streams on a connection. Connection-level flow control works in the same way as stream-level flow control, but the bytes delivered and highest received offset are all aggregates across all streams.

3.5. Authenticated and Encrypted Header and Payload

TCP headers appear in plaintext on the wire and are not authenticated, causing a plethora of injection and header manipulation issues for TCP, such as receive-window manipulation and sequence-number overwriting. While some of these are mechanisms used by middleboxes to improve TCP performance, others are active attacks. Even "performance-enhancing" middleboxes that routinely interpose on the transport state machine end up limiting the evolvability of the transport protocol, as has been observed in the design of MPTCP [RFC6824] and in its subsequent deployability issues.

Generally, QUIC packets are always authenticated and the payload is typically fully encrypted. The parts of the packet header which are not encrypted are still authenticated by the receiver, so as to thwart any packet injection or manipulation by third parties. Some early handshake packets, such as the Version Negotiation packet, are not encrypted, but information sent in these unencrypted handshake packets is later verified as part of cryptographic processing.

PUBLIC_RESET packets that reset a connection are currently not authenticated.

3.6. Connection Migration and Resilience to NAT Rebinding

QUIC connections are identified by a 64-bit Connection ID, randomly generated by the client. QUIC’s consistent connection ID allows connections to survive changes to the client’s IP and port, such as those caused by NAT rebindings or by the client changing network connectivity to a new address. QUIC provides automatic cryptographic verification of a rebound client, since the client continues to use the same session key for encrypting and decrypting packets. The consistent connection ID can be used to allow migration of the connection to a new server IP address as well, since the Connection ID remains consistent across changes in the client’s and the server’s network addresses.

3.7. Version Negotiation

QUIC version negotiation allows for multiple versions of the protocol to be deployed and used concurrently. Version negotiation is described in Section 7.1.

4. Versions

QUIC versions are identified using a 32-bit value.

The version 0x00000000 is reserved to represent an invalid version. This version of the specification is identified by the number 0x00000001.

Versions with the most significant 16 bits of the version number cleared are reserved for use in future IETF consensus documents.

Versions that follow the pattern 0x?a?a?a?a are reserved for use in forcing version negotiation to be exercised. That is, any version number where the low four bits of all octets is 1010 (in binary). A client or server MAY advertise support for any of these reserved versions.

Reserved version numbers will probably never represent a real protocol; a client MAY use one of these version numbers with the expectation that the server will initiate version negotiation; a server MAY advertise support for one of these versions and can expect that clients ignore the value.

[[RFC editor: please remove the remainder of this section before publication.]]

The version number for the final version of this specification (0x00000001), is reserved for the version of the protocol that is published as an RFC.

Version numbers used to identify IETF drafts are created by adding the draft number to 0xff000000. For example, draft-ietf-quic-transport-13 would be identified as 0xff00000D.

Implementors are encouraged to register version numbers of QUIC that they are using for private experimentation on the github wiki [4].

5. Packet Types and Formats

We first describe QUIC’s packet types and their formats, since some are referenced in subsequent mechanisms.

All numeric values are encoded in network byte order (that is, big-endian) and all field sizes are in bits. When discussing individual bits of fields, the least significant bit is referred to as bit 0. Hexadecimal notation is used for describing the value of fields.

Any QUIC packet has either a long or a short header, as indicated by the Header Form bit. Long headers are expected to be used early in the connection before version negotiation and establishment of 1-RTT keys, and for public resets. Short headers are minimal version-specific headers, which can be used after version negotiation and 1-RTT keys are established.

5.1. Long Header

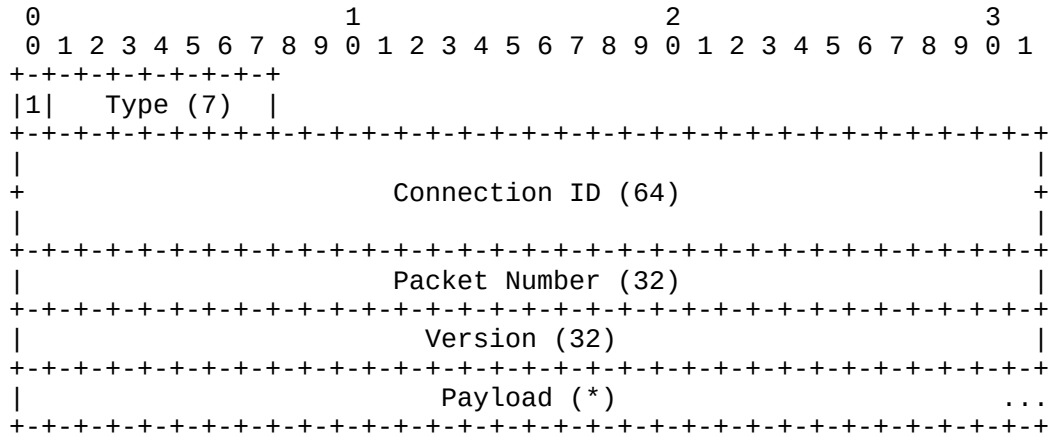


Figure 1: Long Header Format

Long headers are used for packets that are sent prior to the completion of version negotiation and establishment of 1-RTT keys. Once both conditions are met, a sender SHOULD switch to sending short-form headers. While inefficient, long headers MAY be used for packets encrypted with 1-RTT keys. The long form allows for special packets, such as the Version Negotiation and the Public Reset packets to be represented in this uniform fixed-length packet format. A long header contains the following fields:

Header Form: The most significant bit (0x80) of the first octet is set to 1 for long headers and 0 for short headers.

Long Packet Type: The remaining seven bits of first octet of a long packet is the packet type. This field can indicate one of 128 packet types. The types specified for this version are listed in Table 1.

Connection ID: Octets 1 through 8 contain the connection ID. Section 5.7 describes the use of this field in more detail.

Packet Number: Octets 9 to 12 contain the packet number. {{packet-numbers} describes the use of packet numbers.

Version: Octets 13 to 16 contain the selected protocol version. This field indicates which version of QUIC is in use and determines how the rest of the protocol fields are interpreted.

Payload: Octets from 17 onwards (the rest of QUIC packet) are the payload of the packet.

The following packet types are defined:

Type	Name	Section
01	Version Negotiation	Section 5.3
02	Client Cleartext	Section 5.4
03	Non-Final Server Cleartext	Section 5.4
04	Final Server Cleartext	Section 5.4
05	0-RTT Encrypted	Section 5.5
06	1-RTT Encrypted (key phase 0)	Section 5.5
07	1-RTT Encrypted (key phase 1)	Section 5.5
08	Public Reset	Section 5.6

Table 1: Long Header Packet Types

The header form, packet type, connection ID, packet number and version fields of a long header packet are version-independent. The types of packets defined in Table 1 are version-specific. See Section 5.9 for details on how packets from different versions of QUIC are interpreted.

(TODO: Should the list of packet types be version-independent?)

The interpretation of the fields and the payload are specific to a version and packet type. Type-specific semantics for this version

are described in Section 5.3, Section 5.6, Section 5.4, and Section 5.5.

5.2. Short Header

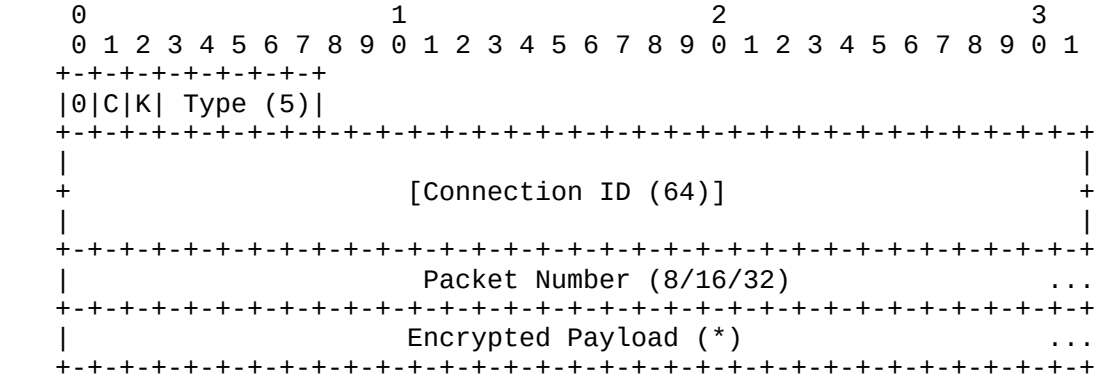


Figure 2: Short Header Format

The short header can be used after the version and 1-RTT keys are negotiated. This header form has the following fields:

Header Form: The most significant bit (0x80) of the first octet of a packet is the header form. This bit is set to 0 for the short header.

Connection ID Flag: The second bit (0x40) of the first octet indicates whether the Connection ID field is present. If set to 1, then the Connection ID field is present; if set to 0, the Connection ID field is omitted.

Key Phase Bit: The third bit (0x20) of the first octet indicates the key phase, which allows a recipient of a packet to identify the packet protection keys that are used to protect the packet. See [QUIC-TLS] for details.

Short Packet Type: The remaining 5 bits of the first octet include one of 32 packet types. Table 2 lists the types that are defined for short packets.

Connection ID: If the Connection ID Flag is set, a connection ID occupies octets 1 through 8 of the packet. See Section 5.7 for more details.

Packet Number: The length of the packet number field depends on the packet type. This field can be 1, 2 or 4 octets long depending on the short packet type.

Encrypted Payload: Packets with a short header always include a 1-RTT protected payload.

The packet type in a short header currently determines only the size of the packet number field. Additional types can be used to signal the presence of other fields.

Type	Packet Number Size
01	1 octet
02	2 octets
03	4 octets

Table 2: Short Header Packet Types

The header form, connection ID flag and connection ID of a short header packet are version-independent. The remaining fields are specific to the selected QUIC version. See Section 5.9 for details on how packets from different versions of QUIC are interpreted.

5.3. Version Negotiation Packet

A Version Negotiation packet is sent only by servers and is a response to a client packet of an unsupported version. It uses a long header and contains:

- Octet 0: 0x81
- Octets 1-8: Connection ID (echoed)
- Octets 9-12: Packet Number (echoed)
- Octets 13-16: Version (echoed)
- Octets 17+: Payload

The payload of the Version Negotiation packet is a list of 32-bit versions which the server supports, as shown below.

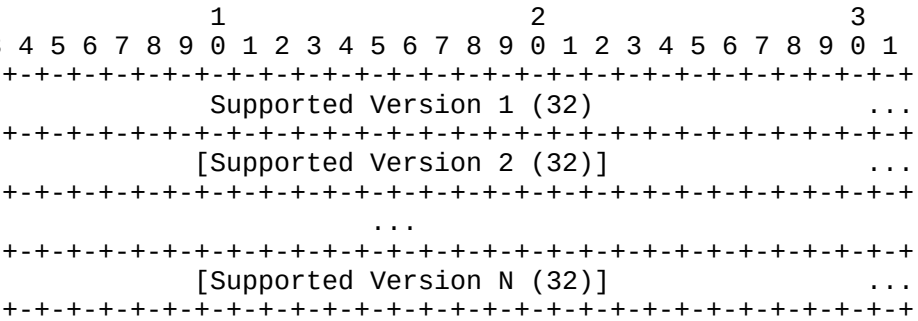


Figure 3: Version Negotiation Packet

See Section 7.1 for a description of the version negotiation process.

5.4. Cleartext Packets

Cleartext packets are sent during the handshake prior to key negotiation. A Client Cleartext packet contains:

- Octet 0: 0x82
- Octets 1-8: Connection ID (initial)
- Octets 9-12: Packet number
- Octets 13-16: Version
- Octets 17+: Payload

Non-Final Server Cleartext packets contain:

- Octet 0: 0x83
- Octets 1-8: Connection ID (echoed)
- Octets 9-12: Packet Number
- Octets 13-16: Version
- Octets 17+: Payload

Final Server Cleartext packets contains:

- Octet 0: 0x84
- Octets 1-8: Connection ID (final)

- Octets 9-12: Packet Number
- Octets 13-16: Version
- Octets 17+: Payload

The client MUST choose a random 64-bit value and use it as the initial Connection ID in all packets until the server replies with the final Connection ID. The server echoes the client's Connection ID in Non-Final Server Cleartext packets. The first Final Server Cleartext and all subsequent packets MUST use the final Connection ID, as described in Section 5.7.

The payload of a Cleartext packet consists of a sequence of frames, as described in Section 6.

(TODO: Add hash before frames.)

5.5. Encrypted Packets

Packets encrypted with either 0-RTT or 1-RTT keys may be sent with long headers. Different packet types explicitly indicate the encryption level for ease of decryption. These packets contain:

- Octet 0: 0x85, 0x86 or 0x87
- Octets 1-8: Connection ID (initial or final)
- Octets 9-12: Packet Number
- Octets 13-16: Version
- Octets 17+: Encrypted Payload

A first octet of 0x85 indicates a 0-RTT packet. After the 1-RTT keys are established, key phases are used by the QUIC packet protection to identify the correct packet protection keys. The initial key phase is 0. See [QUIC-TLS] for more details.

The encrypted payload is both authenticated and encrypted using packet protection keys. [QUIC-TLS] describes packet protection in detail. After decryption, the plaintext consists of a sequence of frames, as described in Section 6.

5.6. Public Reset Packet

A Public Reset packet is only sent by servers and is used to abruptly terminate communications. Public Reset is provided as an option of last resort for a server that does not have access to the state of a connection. This is intended for use by a server that has lost state (for example, through a crash or outage). A server that wishes to communicate a fatal connection error MUST use a CONNECTION_CLOSE frame if it has sufficient state to do so.

A Public Reset packet contains:

- Octet 0: 0x88
- Octets 1-8: Echoed data (octets 1-8 of received packet)
- Octets 9-12: Echoed data (octets 9-12 of received packet)
- Octets 13-16: Version
- Octets 17+: Public Reset Proof

For a client that sends a connection ID on every packet, the Connection ID field is simply an echo of the initial Connection ID, and the Packet Number field includes an echo of the client's packet number (and, depending on the client's packet number length, 0, 2, or 3 additional octets from the client's packet).

A Public Reset packet sent by a server indicates that it does not have the state necessary to continue with a connection. In this case, the server will include the fields that prove that it originally participated in the connection (see Section 5.6.1 for details).

Upon receipt of a Public Reset packet that contains a valid proof, a client MUST tear down state associated with the connection. The client MUST then cease sending packets on the connection and SHOULD discard any subsequent packets that arrive. A Public Reset that does not contain a valid proof MUST be ignored.

5.6.1. Public Reset Proof

TODO: Details to be added.

5.7. Connection ID

QUIC connections are identified by their 64-bit Connection ID. All long headers contain a Connection ID. Short headers indicate the presence of a Connection ID using the CONNECTION_ID flag. When present, the Connection ID is in the same location in all packet headers, making it straightforward for middleboxes, such as load balancers, to locate and use it.

When a connection is initiated, the client MUST choose a random value and use it as the initial Connection ID until the final value is available. The initial Connection ID is a suggestion to the server. The server echoes this value in all packets until the handshake is successful (see [QUIC-TLS]). On a successful handshake, the server MUST select the final Connection ID for the connection and use it in Final Server Cleartext packets. This final Connection ID MAY be the one proposed by the client or MAY be a new server-selected value. All subsequent packets from the server MUST contain this value. On handshake completion, the client MUST switch to using the final Connection ID for all subsequent packets.

Thus, all Client Cleartext packets, 0-RTT Encrypted packets, and Non-Final Server Cleartext packets MUST use the client's randomly-generated initial Connection ID. Final Server Cleartext packets, 1-RTT Encrypted packets, and all short-header packets MUST use the final Connection ID.

5.8. Packet Numbers

The packet number is a 64-bit unsigned number and is used as part of a cryptographic nonce for packet encryption. Each endpoint maintains a separate packet number for sending and receiving. The packet number for sending MUST increase by at least one after sending any packet.

A QUIC endpoint MUST NOT reuse a packet number within the same connection (that is, under the same cryptographic keys). If the packet number for sending reaches 2^64 - 1, the sender MUST close the connection by sending a CONNECTION_CLOSE frame with the error code QUIC_SEQUENCE_NUMBER_LIMIT_REACHED (connection termination is described in Section 7.6.)

To reduce the number of bits required to represent the packet number over the wire, only the least significant bits of the packet number are transmitted over the wire, up to 32 bits. The actual packet number for each packet is reconstructed at the receiver based on the largest packet number received on a successfully authenticated packet.

A packet number is decoded by finding the packet number value that is closest to the next expected packet. The next expected packet is the highest received packet number plus one. For example, if the highest successfully authenticated packet had a packet number of 0xaa82f30e, then a packet containing a 16-bit value of 0x1f94 will be decoded as 0xaa831f94.

The sender MUST use a packet number size able to represent more than twice as large a range than the difference between the largest acknowledged packet and packet number being sent. A peer receiving the packet will then correctly decode the packet number, unless the packet is delayed in transit such that it arrives after many higher-numbered packets have been received. An endpoint MAY use a larger packet number size to safeguard against such reordering.

As a result, the size of the packet number encoding is at least one more than the base 2 logarithm of the number of contiguous unacknowledged packet numbers, including the new packet.

For example, if an endpoint has received an acknowledgment for packet 0x6afa2f, sending a packet with a number of 0x6b4264 requires a 16-bit or larger packet number encoding; whereas a 32-bit packet number is needed to send a packet with a number of 0x6bc107.

5.8.1. Initial Packet Number

The initial value for packet number MUST be a 31-bit random number. That is, the value is selected from an uniform random distribution between 0 and 2^31-1. [RFC4086] provides guidance on the generation of random values.

The first set of packets sent by an endpoint MUST include the low 32-bits of the packet number. Once any packet has been acknowledged, subsequent packets can use a shorter packet number encoding.

5.9. Handling Packets from Different Versions

Between different versions the following things are guaranteed to remain constant:

- the location of the header form flag,
- the location of the Connection ID flag in short headers,
- the location and size of the Connection ID field in both header forms,
- the location and size of the Version field in long headers, and

o the location and size of the Packet Number field in long headers.

Implementations MUST assume that an unsupported version uses an unknown packet format. All other fields MUST be ignored when processing a packet that contains an unsupported version.

6. Frames and Frame Types

The payload of cleartext packets and the plaintext after decryption of encrypted payloads consists of a sequence of frames, as shown in Figure 4.

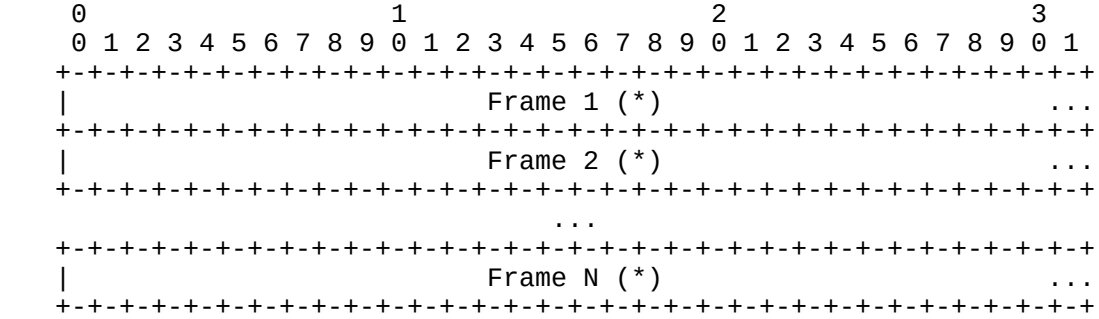


Figure 4: Contents of Encrypted Payload

Encrypted payloads MUST contain at least one frame, and MAY contain multiple frames and multiple frame types.

Frames MUST fit within a single QUIC packet and MUST NOT span a QUIC packet boundary. Each frame begins with a Frame Type byte, indicating its type, followed by additional type-dependent fields:

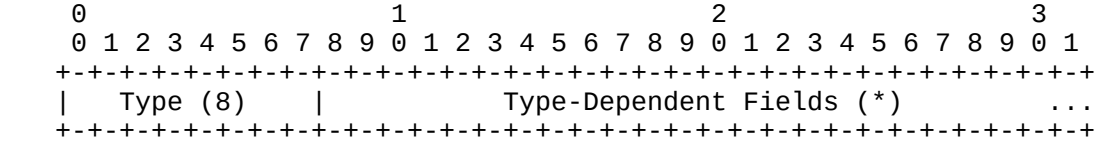


Figure 5: Generic Frame Layout

Frame types are listed in Table 3. Note that the Frame Type byte in STREAM and ACK frames is used to carry other frame-specific flags. For all other frames, the Frame Type byte simply identifies the frame. These frames are explained in more detail as they are referenced later in the document.

Type-field value	Frame type	Definition
0x00	PADDING	Section 8.6
0x01	RST_STREAM	Section 8.5
0x02	CONNECTION_CLOSE	Section 8.8
0x03	GOAWAY	Section 8.9
0x04	WINDOW_UPDATE	Section 8.3
0x05	BLOCKED	Section 8.4
0x07	PING	Section 8.7
0x40 - 0x7f	ACK	Section 8.2
0x80 - 0xff	STREAM	Section 8.1

Table 3: Frame Types

7. Life of a Connection

A QUIC connection is a single conversation between two QUIC endpoints. QUIC's connection establishment intertwines version negotiation with the cryptographic and transport handshakes to reduce connection establishment latency, as described in Section 7.2. Once established, a connection may migrate to a different IP or port at either endpoint, due to NAT rebinding or mobility, as described in Section 7.5. Finally a connection may be terminated by either endpoint, as described in Section 7.6.

7.1. Version Negotiation

QUIC's connection establishment begins with version negotiation, since all communication between the endpoints, including packet and frame formats, relies on the two endpoints agreeing on a version.

A QUIC connection begins with a client sending a handshake packet. The details of the handshake mechanisms are described in Section 7.2, but all of the initial packets sent from the client to the server MUST use the long header format and MUST specify the version of the protocol being used.

When the server receives a packet from a client with the long header format, it compares the client's version to the versions it supports.

If the version selected by the client is not acceptable to the server, the server discards the incoming packet and responds with a Version Negotiation packet (Section 5.3). This includes a list of versions that the server will accept. A server MUST send a Version Negotiation packet for every packet that it receives with an unacceptable version.

If the packet contains a version that is acceptable to the server, the server proceeds with the handshake (Section 7.2). This commits the server to the version that the client selected.

When the client receives a Version Negotiation packet from the server, it should select an acceptable protocol version. If the server lists an acceptable version, the client selects that version and reattempts to create a connection using that version. Though the contents of a packet might not change in response to version negotiation, a client MUST increase the packet number it uses on every packet it sends. Packets MUST continue to use long headers and MUST include the new negotiated protocol version.

The client MUST use the long header format and include its selected version on all packets until it has 1-RTT keys and it has received a packet from the server which is not a Version Negotiation packet.

A client MUST NOT change the version it uses unless it is in response to a Version Negotiation packet from the server. Once a client receives a packet from the server which is not a Version Negotiation packet, it MUST ignore Version Negotiation packets on the same connection.

Version negotiation uses unprotected data. The result of the negotiation MUST be revalidated as part of the cryptographic handshake (see Section 7.3.4).

7.1.1. Using Reserved Versions

For a server to use a new version in the future, clients must correctly handle unsupported versions. To help ensure this, a server SHOULD include a reserved version (see Section 4) while generating a Version Negotiation packet.

The design of version negotiation permits a server to avoid maintaining state for packets that it rejects in this fashion. However, when the server generates a Version Negotiation packet, it cannot randomly generate a reserved version number. This is because

the server is required to include the same value in its transport parameters (see Section 7.3.4). To avoid the selected version number changing during connection establishment, the reserved version SHOULD be generated as a function of values that will be available to the server when later generating its handshake packets.

A pseudorandom function that takes client address information (IP and port) and the client selected version as input would ensure that there is sufficient variability in the values that a server uses.

A client MAY send a packet using a reserved version number. This can be used to solicit a list of supported versions from a server.

7.2. Cryptographic and Transport Handshake

QUIC relies on a combined cryptographic and transport handshake to minimize connection establishment latency. QUIC allocates stream 1 for the cryptographic handshake. This version of QUIC uses TLS 1.3 [QUIC-TLS].

QUIC provides this stream with reliable, ordered delivery of data. In return, the cryptographic handshake provides QUIC with:

- authenticated key exchange, where
  - a server is always authenticated,
  - a client is optionally authenticated,
  - every connection produces distinct and unrelated keys,
  - keying material is usable for packet protection for both 0-RTT and 1-RTT packets, and
  - 1-RTT keys have forward secrecy
- authenticated values for the transport parameters of the peer (see Section 7.3)
- authenticated confirmation of version negotiation (see Section 7.3.4)
- authenticated negotiation of an application protocol (TLS uses ALPN [RFC7301] for this purpose)
- for the server, the ability to carry data that provides assurance that the client can receive packets that are addressed with the transport address that is claimed by the client (see Section 7.4)

The initial cryptographic handshake message MUST be sent in a single packet. Any second attempt that is triggered by address validation MUST also be sent within a single packet. This avoids having to reassemble a message from multiple packets. Reassembling messages requires that a server maintain state prior to establishing a connection, exposing the server to a denial of service risk.

The first client packet of the cryptographic handshake protocol MUST fit within a 1280 octet QUIC packet. This includes overheads that reduce the space available to the cryptographic handshake protocol.

Details of how TLS is integrated with QUIC is provided in more detail in [QUIC-TLS].

7.3. Transport Parameters

During connection establishment, both endpoints make authenticated declarations of their transport parameters. These declarations are made unilaterally by each endpoint. Endpoints are required to comply with the restrictions implied by these parameters; the description of each parameter includes rules for its handling.

The format of the transport parameters is the TransportParameters struct from Figure 6. This is described using the presentation language from Section 3 of [I-D.ietf-tls-tls13].

```
uint32 QuicVersion;

enum {
   stream_fc_offset(0),
   connection_fc_offset(1),
   concurrent_streams(2),
   idle_timeout(3),
   truncate_connection_id(4),
   (65535)
} TransportParameterId;

struct {
   TransportParameterId parameter;
   opaque value<0..2^16-1>;
} TransportParameter;

struct {
   select (Handshake.msg_type) {
      case client_hello:
         QuicVersion negotiated_version;
         QuicVersion initial_version;

      case encrypted_extensions:
         QuicVersion supported_versions<2..2^8-4>;
   };
   TransportParameter parameters<30..2^16-1>;
} TransportParameters;
```

Figure 6: Definition of TransportParameters

The "extension_data" field of the quic_transport_parameters extension defined in [QUIC-TLS] contains a TransportParameters value. TLS encoding rules are therefore used to encode the transport parameters.

QUIC encodes transport parameters into a sequence of octets, which are then included in the cryptographic handshake. Once the handshake completes, the transport parameters declared by the peer are available. Each endpoint validates the value provided by its peer. In particular, version negotiation MUST be validated (see Section 7.3.4) before the connection establishment is considered properly complete.

Definitions for each of the defined transport parameters are included in Section 7.3.1.

7.3.1. Transport Parameter Definitions

An endpoint MUST include the following parameters in its encoded TransportParameters:

stream_fc_offset (0x0000): The initial stream level flow control offset parameter is encoded as an unsigned 32-bit integer in units of octets. The sender of this parameter indicates that the flow control offset for all stream data sent toward it is this value.

connection_fc_offset (0x0001): The connection level flow control offset parameter contains the initial connection flow control window encoded as an unsigned 32-bit integer in units of 1024 octets. That is, the value here is multiplied by 1024 to determine the actual flow control offset. The sender of this parameter sets the byte offset for connection level flow control to this value. This is equivalent to sending a WINDOW_UPDATE (Section 8.3) for the connection immediately after completing the handshake.

concurrent_streams (0x0002): The maximum number of concurrent streams parameter is encoded as an unsigned 32-bit integer.

idle_timeout (0x0003): The idle timeout is a value in seconds that is encoded as an unsigned 16-bit integer. The maximum value is 600 seconds (10 minutes).

An endpoint MAY use the following transport parameters:

truncate_connection_id (0x0004): The truncated connection identifier parameter indicates that packets sent to the peer can omit the connection ID. This can be used by an endpoint where the 5-tuple is sufficient to identify a connection. This parameter is zero length. Omitting the parameter indicates that the endpoint relies on the connection ID being present in every packet.

7.3.2. Values of Transport Parameters for 0-RTT

Transport parameters from the server SHOULD be remembered by the client for use with 0-RTT data. A client that doesn’t remember values from a previous connection can instead assume the following values: stream_fc_offset (65535), connection_fc_offset (65535), concurrent_streams (10), idle_timeout (600), truncate_connection_id (absent).

If assumed values change as a result of completing the handshake, the client is expected to respect the new values. This introduces some

potential problems, particularly with respect to transport parameters that establish limits:

- A client might exceed a newly declared connection or stream flow control limit with 0-RTT data. If this occurs, the client ceases transmission as though the flow control limit was reached. Once WINDOW_UPDATE frames indicating an increase to the affected flow control offsets is received, the client can recommence sending.

- Similarly, a client might exceed the concurrent stream limit declared by the server. A client MUST reset any streams that exceed this limit. A server SHOULD reset any streams it cannot handle with a code that allows the client to retry any application action bound to those streams.

A server MAY close a connection if remembered or assumed 0-RTT transport parameters cannot be supported, using an error code that is appropriate to the specific condition. For example, a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA might be used to indicate that exceeding flow control limits caused the error. A client that has a connection closed due to an error condition SHOULD NOT attempt 0-RTT when attempting to create a new connection.

7.3.3. New Transport Parameters

New transport parameters can be used to negotiate new protocol behavior. An endpoint MUST ignore transport parameters that it does not support. Absence of a transport parameter therefore disables any optional protocol feature that is negotiated using the parameter.

The definition of a transport parameter SHOULD include a default value that a client can use when establishing a new connection. If no default is specified, the value can be assumed to be absent when attempting 0-RTT.

New transport parameters can be registered according to the rules in Section 14.1.

7.3.4. Version Negotiation Validation

The transport parameters include three fields that encode version information. These retroactively authenticate the version negotiation (see Section 7.1) that is performed prior to the cryptographic handshake.

The cryptographic handshake provides integrity protection for the negotiated version as part of the transport parameters (see

Section 7.3). As a result, modification of version negotiation packets by an attacker can be detected.

The client includes two fields in the transport parameters:

- The negotiated_version is the version that was finally selected for use. This MUST be identical to the value that is on the packet that carries the ClientHello. A server that receives a negotiated_version that does not match the version of QUIC that is in use MUST terminate the connection with a QUIC_VERSION_NEGOTIATION_MISMATCH error code.

- The initial_version is the version that the client initially attempted to use. If the server did not send a version negotiation packet Section 5.3, this will be identical to the negotiated_version.

A server that processes all packets in a stateful fashion can remember how version negotiation was performed and validate the initial_version value.

A server that does not maintain state for every packet it receives (i.e., a stateless server) uses a different process. If the initial and negotiated versions are the same, a stateless server can accept the value.

If the initial version is different from the negotiated_version, a stateless server MUST check that it would have sent a version negotiation packet if it had received a packet with the indicated initial_version. If a server would have accepted the version included in the initial_version and the value differs from the value of negotiated_version, the server MUST terminate the connection with a QUIC_VERSION_NEGOTIATION_MISMATCH error.

The server includes a list of versions that it would send in any version negotiation packet (Section 5.3) in supported_versions. This value is set even if it did not send a version negotiation packet.

The client can validate that the negotiated_version is included in the supported_versions list and - if version negotiation was performed - that it would have selected the negotiated version. A client MUST terminate the connection with a QUIC_VERSION_NEGOTIATION_MISMATCH error code if the negotiated_version value is not included in the supported_versions list. A client MUST terminate with a QUIC_VERSION_NEGOTIATION_MISMATCH error code if version negotiation occurred but it would have selected a different version based on the value of the supported_versions list.

7.4. Proof of Source Address Ownership

Transport protocols commonly spend a round trip checking that a client owns the transport address (IP and port) that it claims. Verifying that a client can receive packets sent to its claimed transport address protects against spoofing of this information by malicious clients.

This technique is used primarily to avoid QUIC from being used for traffic amplification attack. In such an attack, a packet is sent to a server with spoofed source address information that identifies a victim. If a server generates more or larger packets in response to that packet, the attacker can use the server to send more data toward the victim than it would be able to send on its own.

Several methods are used in QUIC to mitigate this attack. Firstly, the initial handshake packet from a client is padded to at least 1280 octets. This allows a server to send a similar amount of data without risking causing an amplication attack toward an unproven remote address.

A server eventually confirms that a client has received its messages when the cryptographic handshake successfully completes. This might be insufficient, either because the server wishes to avoid the computational cost of completing the handshake, or it might be that the size of the packets that are sent during the handshake is too large. This is especially important for 0-RTT, where the server might wish to provide application data traffic - such as a response to a request - in response to the data carried in the early data from the client.

To send additional data prior to completing the cryptographic handshake, the server then needs to validate that the client owns the address that it claims.

Source address validation is therefore performed during the establishment of a connection. TLS provides the tools that support the feature, but basic validation is performed by the core transport protocol.

7.4.1. Client Address Validation Procedure

QUIC uses token-based address validation. Any time the server wishes to validate a client address, it provides the client with a token. As long as the token cannot be easily guessed (see Section 7.4.3), if the client is able to return that token, it proves to the server that it received the token.

During the processing of the cryptographic handshake messages from a client, TLS will request that QUIC make a decision about whether to proceed based on the information it has. TLS will provide QUIC with any token that was provided by the client. For an initial packet, QUIC can decide to abort the connection, allow it to proceed, or request address validation.

If QUIC decides to request address validation, it provides the cryptographic handshake with a token. The contents of this token are consumed by the server that generates the token, so there is no need for a single well-defined format. A token could include information about the claimed client address (IP and port), a timestamp, and any other supplementary information the server will need to validate the token in the future.

The cryptographic handshake is responsible for enacting validation by sending the address validation token to the client. A legitimate client will include a copy of the token when it attempts to continue the handshake. The cryptographic handshake extracts the token then asks QUIC a second time whether the token is acceptable. In response, QUIC can either abort the connection or permit it to proceed.

A connection MAY be accepted without address validation - or with only limited validation - but a server SHOULD limit the data it sends toward an unvalidated address. Successful completion of the cryptographic handshake implicitly provides proof that the client has received packets from the server.

7.4.2. Address Validation on Session Resumption

A server MAY provide clients with an address validation token during one connection that can be used on a subsequent connection. Address validation is especially important with 0-RTT because a server potentially sends a significant amount of data to a client in response to 0-RTT data.

A different type of token is needed when resuming. Unlike the token that is created during a handshake, there might be some time between when the token is created and when the token is subsequently used. Thus, a resumption token SHOULD include an expiration time. It is also unlikely that the client port number is the same on two different connections; validating the port is therefore unlikely to be successful.

This token can be provided to the cryptographic handshake immediately after establishing a connection. QUIC might also generate an updated token if significant time passes or the client address changes for

any reason (see Section 7.5). The cryptographic handshake is responsible for providing the client with the token. In TLS the token is included in the ticket that is used for resumption and 0-RTT, which is carried in a NewSessionTicket message.

7.4.3. Address Validation Token Integrity

An address validation token MUST be difficult to guess. Including a large enough random value in the token would be sufficient, but this depends on the server remembering the value it sends to clients.

A token-based scheme allows the server to offload any state associated with validation to the client. For this design to work, the token MUST be covered by integrity protection against modification or falsification by clients. Without integrity protection, malicious clients could generate or guess values for tokens that would be accepted by the server. Only the server requires access to the integrity protection key for tokens.

In TLS the address validation token is often bundled with the information that TLS requires, such as the resumption secret. In this case, adding integrity protection can be delegated to the cryptographic handshake protocol, avoiding redundant protection. If integrity protection is delegated to the cryptographic handshake, an integrity failure will result in immediate cryptographic handshake failure. If integrity protection is performed by QUIC, QUIC MUST abort the connection if the integrity check fails with a QUIC_ADDRESS_VALIDATION_FAILURE error code.

7.5. Connection Migration

QUIC connections are identified by their 64-bit Connection ID. QUIC's consistent connection ID allows connections to survive changes to the client's IP and/or port, such as those caused by client or server migrating to a new network. QUIC also provides automatic cryptographic verification of a client which has changed its IP address because the client continues to use the same session key for encrypting and decrypting packets.

DISCUSS: Simultaneous migration. Is this reasonable?

TODO: Perhaps move mitigation techniques from Security Considerations here.

7.6. Connection Termination

Connections should remain open until they become idle for a pre-negotiated period of time. A QUIC connection, once established, can be terminated in one of three ways:

1. Explicit Shutdown: An endpoint sends a CONNECTION_CLOSE frame to initiate a connection termination. An endpoint may send a GOAWAY frame to the peer prior to a CONNECTION_CLOSE to indicate that the connection will soon be terminated. A GOAWAY frame signals to the peer that any active streams will continue to be processed, but the sender of the GOAWAY will not initiate any additional streams and will not accept any new incoming streams. On termination of the active streams, a CONNECTION_CLOSE may be sent. If an endpoint sends a CONNECTION_CLOSE frame while unterminated streams are active (no FIN bit or RST_STREAM frames have been sent or received for one or more streams), then the peer must assume that the streams were incomplete and were abnormally terminated.

2. Implicit Shutdown: The default idle timeout for a QUIC connection is 30 seconds, and is a required parameter in connection negotiation. The maximum is 10 minutes. If there is no network activity for the duration of the idle timeout, the connection is closed. By default a CONNECTION_CLOSE frame will be sent. A silent close option can be enabled when it is expensive to send an explicit close, such as mobile networks that must wake up the radio.

3. Abrupt Shutdown: An endpoint may send a Public Reset packet at any time during the connection to abruptly terminate an active connection. A Public Reset packet SHOULD only be used as a final recourse. Commonly, a public reset is expected to be sent when a packet on an established connection is received by an endpoint that is unable decrypt the packet. For instance, if a server reboots mid-connection and loses any cryptographic state associated with open connections, and then receives a packet on an open connection, it should send a Public Reset packet in return. (TODO: articulate rules around when a public reset should be sent.)

TODO: Connections that are terminated are added to a TIME_WAIT list at the server, so as to absorb any straggler packets in the network. Discuss TIME_WAIT list.

8. Frame Types and Formats

As described in Section 6, Regular packets contain one or more frames. We now describe the various QUIC frame types that can be present in a Regular packet. The use of these frames and various frame header bits are described in subsequent sections.

8.1. STREAM Frame

STREAM frames implicitly create a stream and carry stream data. The type byte for a STREAM frame contains embedded flags, and is formatted as "1FDOOOSS". These bits are parsed as follows:

- The leftmost bit must be set to 1, indicating that this is a STREAM frame.
- "F" is the FIN bit, which is used for stream termination.
- The "D" bit indicates whether a Data Length field is present in the STREAM header. When set to 0, this field indicates that the Stream Data field extends to the end of the packet. When set to 1, this field indicates that Data Length field contains the length (in bytes) of the Stream Data field. The option to omit the length should only be used when the packet is a "full-sized" packet, to avoid the risk of corruption via padding.
- The "OOO" bits encode the length of the Offset header field as 0, 16, 24, 32, 40, 48, 56, or 64 bits long.
- The "SS" bits encode the length of the Stream ID header field as 8, 16, 24, or 32 bits. (DISCUSS: Consider making this 8, 16, 32, 64.)

A STREAM frame is shown below.

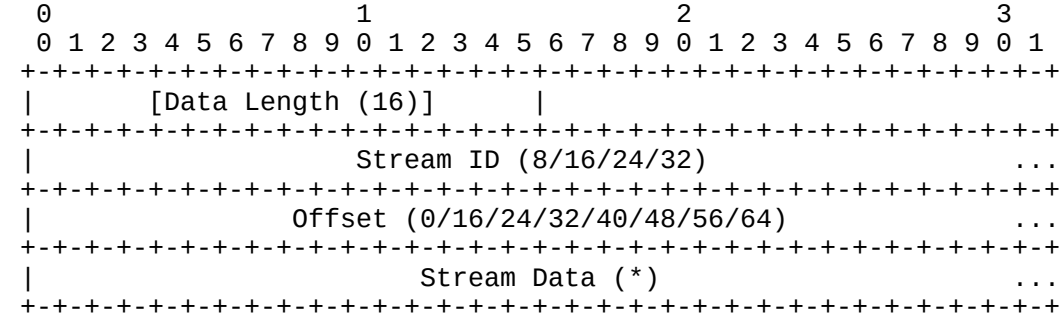


Figure 7: STREAM Frame Format

The STREAM frame contains the following fields:

Data Length: An optional 16-bit unsigned number specifying the length of the Stream Data field in this STREAM frame. This field is present when the "D" bit is set to 1.

Stream ID: A variable-sized unsigned ID unique to this stream.

Offset: A variable-sized unsigned number specifying the byte offset in the stream for the data in this STREAM frame. The first byte in the stream has an offset of 0. The largest offset delivered on a stream - the sum of the re-constructed offset and data length - MUST be less than 2^64.

Stream Data: The bytes from the designated stream to be delivered.

A STREAM frame MUST have either non-zero data length or the FIN bit set.

Stream multiplexing is achieved by interleaving STREAM frames from multiple streams into one or more QUIC packets. A single QUIC packet MAY bundle STREAM frames from multiple streams.

Implementation note: One of the benefits of QUIC is avoidance of head-of-line blocking across multiple streams. When a packet loss occurs, only streams with data in that packet are blocked waiting for a retransmission to be received, while other streams can continue making progress. Note that when data from multiple streams is bundled into a single QUIC packet, loss of that packet blocks all those streams from making progress. An implementation is therefore advised to bundle as few streams as necessary in outgoing packets without losing transmission efficiency to underfilled packets.

8.2. ACK Frame

Receivers send ACK frames to inform senders which packets they have received and processed, as well as which packets are considered missing. The ACK frame contains between 1 and 256 ACK blocks. ACK blocks are ranges of acknowledged packets.

To limit ACK blocks to those that have not yet been received by the sender, the receiver SHOULD track which ACK frames have been acknowledged by its peer. Once an ACK frame has been acknowledged, the packets it acknowledges SHOULD not be acknowledged again. To handle cases where the receiver is only sending ACK frames, and hence will not receive acknowledgments for its packets, it MAY send a PING frame at most once per RTT to explicitly request acknowledgment.

To limit receiver state or the size of ACK frames, a receiver MAY limit the number of ACK blocks it sends. A receiver can do this even without receiving acknowledgment of its ACK frames, with the knowledge this could cause the sender to unnecessarily retransmit some data.

Unlike TCP SACKs, QUIC ACK blocks are cumulative and therefore irrevocable. Once a packet has been acknowledged, even if it does not appear in a future ACK frame, it is assumed to be acknowledged.

QUIC ACK frames contain a timestamp section with up to 255 timestamps. Timestamps enable better congestion control, but are not required for correct loss recovery, and old timestamps are less valuable, so it is not guaranteed every timestamp will be received by the sender. A receiver SHOULD send a timestamp exactly once for each received packet containing retransmittable frames. A receiver MAY send timestamps for non-retransmittable packets.

A sender MAY intentionally skip packet numbers to introduce entropy into the connection, to avoid opportunistic acknowledgement attacks. The sender MUST close the connection if an unsent packet number is acknowledged. The format of the ACK frame is efficient at expressing blocks of missing packets; skipping packet numbers between 1 and 255 effectively provides up to 8 bits of efficient entropy on demand, which should be adequate protection against most opportunistic acknowledgement attacks.

The type byte for a ACK frame contains embedded flags, and is formatted as "01NULLMM". These bits are parsed as follows:

- The first two bits must be set to 01 indicating that this is an ACK frame.
- The "N" bit indicates whether the frame has more than 1 range of acknowledged packets (i.e., whether the ACK Block Section contains a Num Blocks field).
- The "U" bit is unused and MUST be set to zero.
- The two "LL" bits encode the length of the Largest Acknowledged field as 1, 2, 4, or 6 bytes long.
- The two "MM" bits encode the length of the ACK Block Length fields as 1, 2, 4, or 6 bytes long.

An ACK frame is shown below.

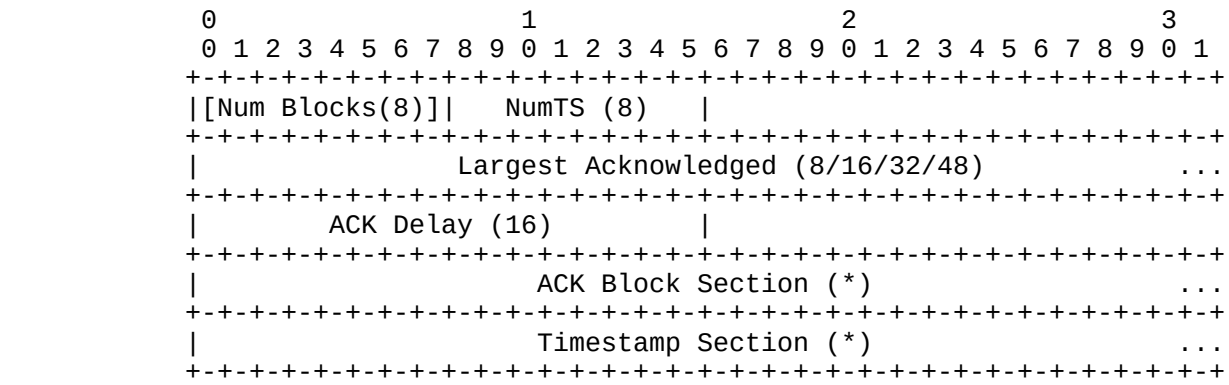


Figure 8: ACK Frame Format

The fields in the ACK frame are as follows:

Num Blocks (opt): An optional 8-bit unsigned value specifying the number of additional ACK blocks (besides the required First ACK Block) in this ACK frame. Only present if the 'N' flag bit is 1.

Num Timestamps: An unsigned 8-bit number specifying the total number of <packet number, timestamp> pairs in the Timestamp Section.

Largest Acknowledged: A variable-sized unsigned value representing the largest packet number the peer is acknowledging in this packet (typically the largest that the peer has seen thus far.)

ACK Delay: The time from when the largest acknowledged packet, as indicated in the Largest Acknowledged field, was received by this peer to when this ACK was sent.

ACK Block Section: Contains one or more blocks of packet numbers which have been successfully received, see Section 8.2.1.

Timestamp Section: Contains zero or more timestamps reporting transit delay of received packets. See Section 8.2.2.

8.2.1. ACK Block Section

The ACK Block Section contains between one and 256 blocks of packet numbers which have been successfully received. If the Num Blocks field is absent, only the First ACK Block length is present in this section. Otherwise, the Num Blocks field indicates how many additional blocks follow the First ACK Block Length field.

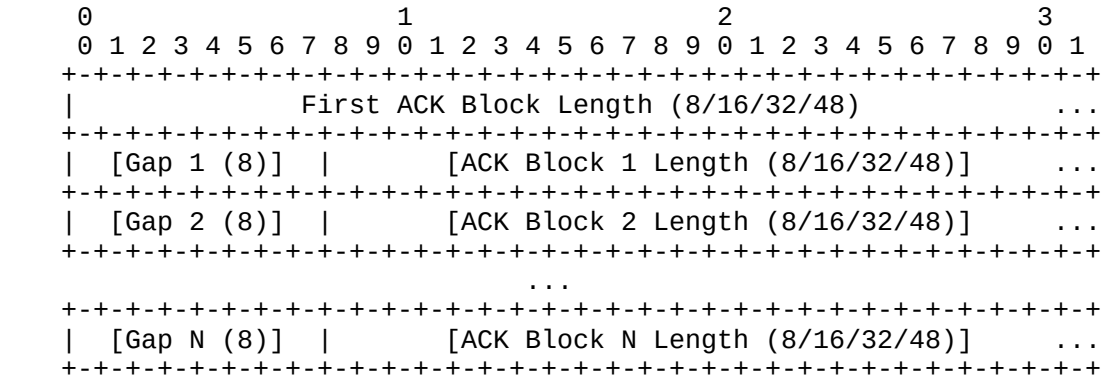


Figure 9: ACK Block Section

The fields in the ACK Block Section are:

First ACK Block Length: An unsigned packet number delta that indicates the number of contiguous additional packets being acknowledged starting at the Largest Acknowledged.

Gap To Next Block (opt, repeated): An unsigned number specifying the number of contiguous missing packets from the end of the previous ACK block to the start of the next. Repeated "Num Blocks" times.

ACK Block Length (opt, repeated): An unsigned packet number delta that indicates the number of contiguous packets being acknowledged starting after the end of the previous gap. Repeated "Num Blocks" times.

8.2.2. Timestamp Section

The Timestamp Section contains between zero and 255 measurements of packet receive times relative to the beginning of the connection.

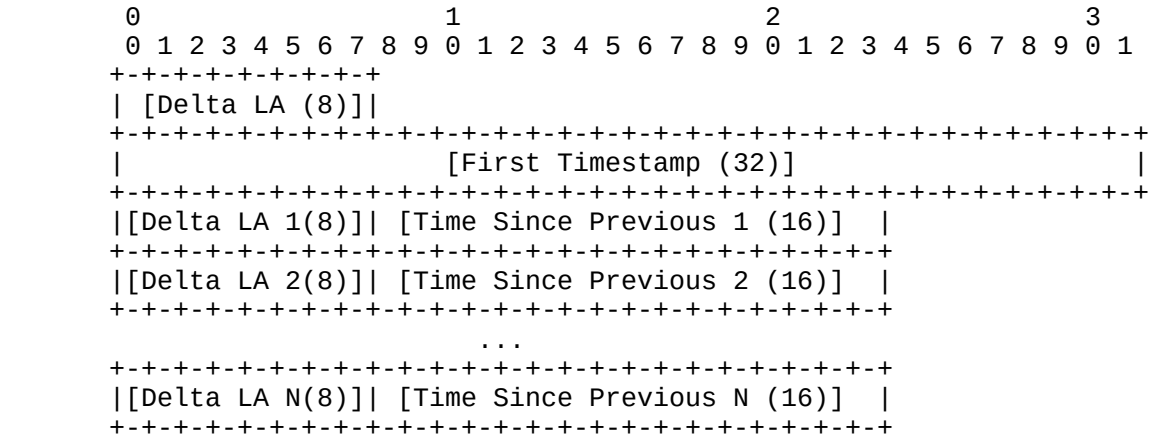


Figure 10: Timestamp Section

The fields in the Timestamp Section are:

Delta Largest Acknowledged (opt): An optional 8-bit unsigned packet number delta specifying the delta between the largest acknowledged and the first packet whose timestamp is being reported. In other words, this first packet number may be computed as (Largest Acknowledged - Delta Largest Acknowledged.)

First Timestamp (opt): An optional 32-bit unsigned value specifying the time delta in microseconds, from the beginning of the connection to the arrival of the packet indicated by Delta Largest Acknowledged.

Delta Largest Acked 1..N (opt, repeated): This field has the same semantics and format as "Delta Largest Acknowledged". Repeated "Num Timestamps - 1" times.

Time Since Previous Timestamp 1..N(opt, repeated): An optional 16-bit unsigned value specifying time delta from the previous reported timestamp. It is encoded in the same format as the ACK Delay. Repeated "Num Timestamps - 1" times.

The timestamp section lists packet receipt timestamps ordered by timestamp.

8.2.2.1. Time Format

DISCUSS_AND_REPLACE: Perhaps make this format simpler.

The time format used in the ACK frame above is a 16-bit unsigned float with 11 explicit bits of mantissa and 5 bits of explicit exponent, specifying time in microseconds. The bit format is loosely modeled after IEEE 754. For example, 1 microsecond is represented as 0x1, which has an exponent of zero, presented in the 5 high order bits, and mantissa of 1, presented in the 11 low order bits. When the explicit exponent is greater than zero, an implicit high-order 12th bit of 1 is assumed in the mantissa. For example, a floating value of 0x800 has an explicit exponent of 1, as well as an explicit mantissa of 0, but then has an effective mantissa of 4096 (12th bit is assumed to be 1). Additionally, the actual exponent is one-less than the explicit exponent, and the value represents 4096 microseconds. Any values larger than the representable range are clamped to 0xFFFF.

8.2.3. ACK Frames and Packet Protection

ACK frames that acknowledge protected packets MUST be carried in a packet that has an equivalent or greater level of packet protection.

Packets that are protected with 1-RTT keys MUST be acknowledged in packets that are also protected with 1-RTT keys.

A packet that is not protected and claims to acknowledge a packet number that was sent with packet protection is not valid. An unprotected packet that carries acknowledgments for protected packets MUST be discarded in its entirety.

Packets that a client sends with 0-RTT packet protection MUST be acknowledged by the server in packets protected by 1-RTT keys. This can mean that the client is unable to use these acknowledgments if the server cryptographic handshake messages are delayed or lost. Note that the same limitation applies to other data sent by the server protected by the 1-RTT keys.

Unprotected packets, such as those that carry the initial cryptographic handshake messages, MAY be acknowledged in unprotected packets. Unprotected packets are vulnerable to falsification or modification. Unprotected packets can be acknowledged along with protected packets in a protected packet.

An endpoint SHOULD acknowledge packets containing cryptographic handshake messages in the next unprotected packet that it sends, unless it is able to acknowledge those packets in later packets protected by 1-RTT keys. At the completion of the cryptographic handshake, both peers send unprotected packets containing cryptographic handshake messages followed by packets protected by 1-RTT keys. An endpoint SHOULD acknowledge the unprotected packets

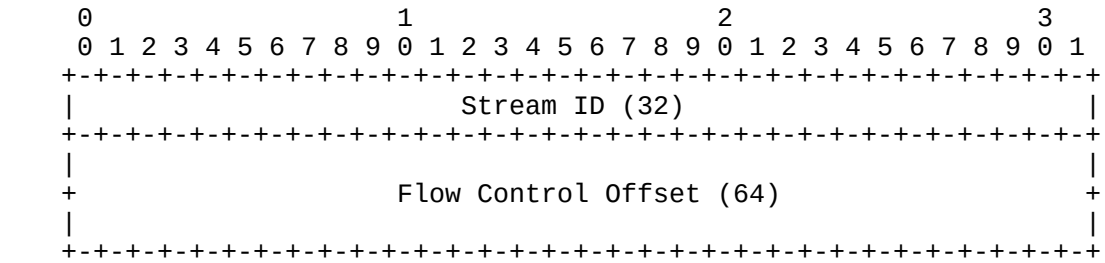
that complete the cryptographic handshake in a protected packet, because its peer is guaranteed to have access to 1-RTT packet protection keys.

For instance, a server acknowledges a TLS ClientHello in the packet that carries the TLS ServerHello; similarly, a client can acknowledge a TLS HelloRetryRequest in the packet containing a second TLS ClientHello. The complete set of server handshake messages (TLS ServerHello through to Finished) might be acknowledged by a client in protected packets, because it is certain that the server is able to decipher the packet.

8.3. WINDOW_UPDATE Frame

The WINDOW_UPDATE frame (type=0x04) informs the peer of an increase in an endpoint's flow control receive window for either a single stream, or the entire connection as a whole.

The frame is as follows:



The fields in the WINDOW_UPDATE frame are as follows:

Stream ID: ID of the stream whose flow control windows is being updated, or 0 to specify the connection-level flow control window.

Flow Control Offset: A 64-bit unsigned integer indicating the flow control offset for the given stream (for a stream ID other than 0) or the entire connection.

The flow control offset is expressed in units of octets for individual streams (for stream identifiers other than 0).

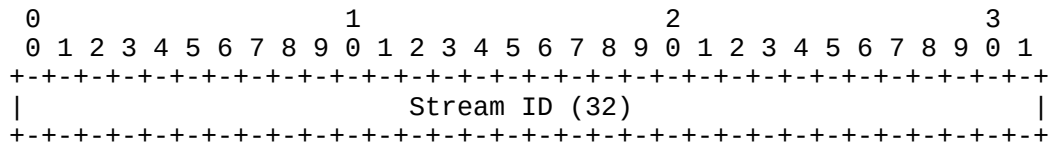
The connection-level flow control offset is expressed in units of 1024 octets (for a stream identifier of 0). That is, the connection-level flow control offset is determined by multiplying the encoded value by 1024.

An endpoint accounts for the maximum offset of data that is sent or received on a stream. Loss or reordering can mean that the maximum offset is greater than the total size of data received on a stream. Similarly, receiving STREAM frames might not increase the maximum offset on a stream. A STREAM frame with a FIN bit set or RST_STREAM causes the final offset for a stream to be fixed.

The maximum data offset on a stream MUST NOT exceed the stream flow control offset advertised by the receiver. The sum of the maximum data offsets of all streams (including closed streams) MUST NOT exceed the connection flow control offset advertised by the receiver. An endpoint MUST terminate a connection with a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA error if it receives more data than the largest flow control offset that it has sent, unless this is a result of a change in the initial offsets (see Section 7.3.2).

8.4. BLOCKED Frame

A sender sends a BLOCKED frame (type=0x05) when it is ready to send data (and has data to send), but is currently flow control blocked. BLOCKED frames are purely informational frames, but extremely useful for debugging purposes. A receiver of a BLOCKED frame should simply discard it (after possibly printing a helpful log message). The frame is as follows:

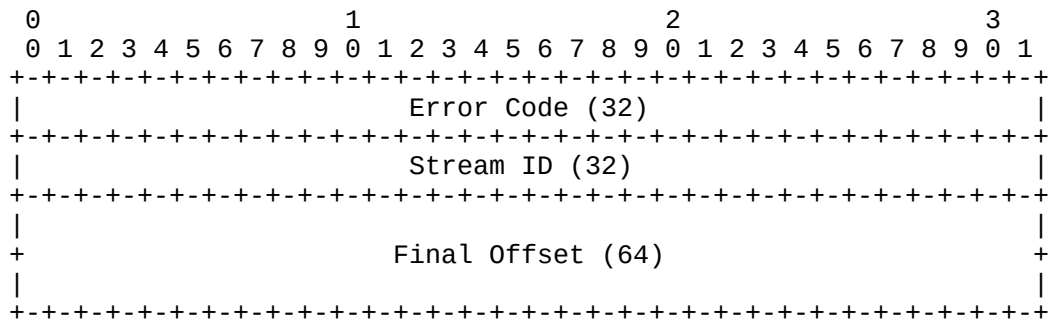


The BLOCKED frame contains a single field:

Stream ID: A 32-bit unsigned number indicating the stream which is flow control blocked. A non-zero Stream ID field specifies the stream that is flow control blocked. When zero, the Stream ID field indicates that the connection is flow control blocked.

8.5. RST_STREAM Frame

An endpoint may use a RST_STREAM frame (type=0x01) to abruptly terminate a stream. The frame is as follows:



The fields are:

Error code: A 32-bit error code which indicates why the stream is being closed.

Stream ID: The 32-bit Stream ID of the stream being terminated.

Final offset: A 64-bit unsigned integer indicating the absolute byte offset of the end of data written on this stream by the RST_STREAM sender.

8.6. PADDING Frame

The PADDING frame (type=0x00) has no semantic value. PADDING frames can be used to increase the size of a packet. Padding can be used to increase an initial client packet to the minimum required size, or to provide protection against traffic analysis for protected packets.

A PADDING frame has no content. That is, a PADDING frame consists of the single octet that identifies the frame as a PADDING frame.

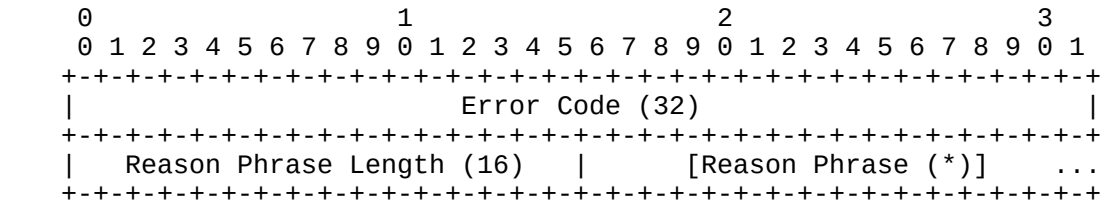
8.7. PING frame

Endpoints can use PING frames (type=0x07) to verify that their peers are still alive or to check reachability to the peer. The PING frame contains no additional fields. The receiver of a PING frame simply needs to acknowledge the packet containing this frame. The PING frame SHOULD be used to keep a connection alive when a stream is open. The default is to send a PING frame after 15 seconds of quiescence. A PING frame has no additional fields.

8.8. CONNECTION_CLOSE frame

An endpoint sends a CONNECTION_CLOSE frame (type=0x02) to notify its peer that the connection is being closed. If there are open streams that haven’t been explicitly closed, they are implicitly closed when

the connection is closed. (Ideally, a GOAWAY frame would be sent with enough time that all streams are torn down.) The frame is as follows:



The fields of a CONNECTION_CLOSE frame are as follows:

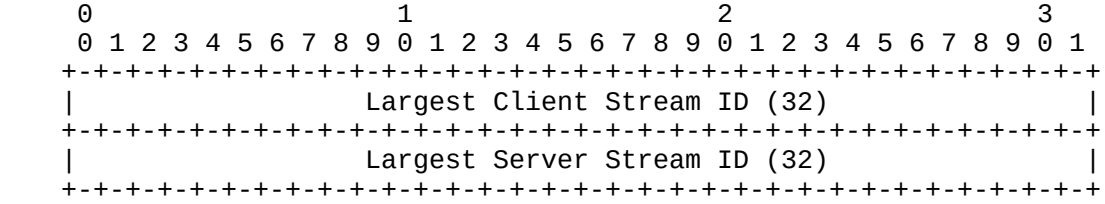
Error Code: A 32-bit error code which indicates the reason for closing this connection.

Reason Phrase Length: A 16-bit unsigned number specifying the length of the reason phrase. This may be zero if the sender chooses to not give details beyond the Error Code.

Reason Phrase: An optional human-readable explanation for why the connection was closed.

8.9. GOAWAY Frame

An endpoint uses a GOAWAY frame (type=0x03) to initiate a graceful shutdown of a connection. The endpoints will continue to use any active streams, but the sender of the GOAWAY will not initiate or accept any additional streams beyond those indicated. The GOAWAY frame is as follows:



The fields of a GOAWAY frame are:

Largest Client Stream ID: The highest-numbered, client-initiated stream on which the endpoint sending the GOAWAY frame either sent data, or received and delivered data. All higher-numbered, client-initiated streams (that is, odd-numbered streams) are implicitly reset by sending or receiving the GOAWAY frame.

Largest Server Stream ID: The highest-numbered, server-initiated stream on which the endpoint sending the GOAWAY frame either sent data, or received and delivered data. All higher-numbered, server-initiated streams (that is, even-numbered streams) are implicitly reset by sending or receiving the GOAWAY frame.

A GOAWAY frame indicates that any application layer actions on streams with higher numbers than those indicated can be safely retried because no data was exchanged. An endpoint MUST set the value of the Largest Client or Server Stream ID to be at least as high as the highest-numbered stream on which it either sent data or received and delivered data to the application protocol that uses QUIC.

An endpoint MAY choose a larger stream identifier if it wishes to allow for a number of streams to be created. This is especially valuable for peer-initiated streams where packets creating new streams could be in transit; using a larger stream number allows those streams to complete.

In addition to initiating a graceful shutdown of a connection, GOAWAY MAY be sent immediately prior to sending a CONNECTION_CLOSE frame that is sent as a result of detecting a fatal error. Higher-numbered streams than those indicated in the GOAWAY frame can then be retried.

9. Packetization and Reliability

The Path Maximum Transmission Unit (PTMU) is the maximum size of the entire IP header, UDP header, and UDP payload. The UDP payload includes the QUIC public header, encrypted payload, and any authentication fields.

All QUIC packets SHOULD be sized to fit within the estimated PMTU to avoid IP fragmentation or packet drops. To optimize bandwidth efficiency, endpoints SHOULD use Packetization Layer PMTU Discovery ([RFC4821]) and MAY use PMTU Discovery ([RFC1191], [RFC1981]) for detecting the PMTU, setting the PMTU appropriately, and storing the result of previous PMTU determinations.

In the absence of these mechanisms, QUIC endpoints SHOULD NOT send IP packets larger than 1280 octets. Assuming the minimum IP header size, this results in a UDP payload length of 1232 octets for IPv6 and 1252 octets for IPv4.

QUIC endpoints that implement any kind of PMTU discovery SHOULD maintain an estimate for each combination of local and remote IP addresses (as each pairing could have a different maximum MTU in the path).

QUIC depends on the network path supporting a MTU of at least 1280 octets. This is the IPv6 minimum and therefore also supported by most modern IPv4 networks. An endpoint MUST NOT reduce their MTU below this number, even if it receives signals that indicate a smaller limit might exist.

Clients MUST ensure that the first packet in a connection, and any retransmissions of those octets, has a total size (including IP and UDP headers) of at least 1280 bytes. This might require inclusion of PADDING frames. It is RECOMMENDED that a packet be padded to exactly 1280 octets unless the client has a reasonable assurance that the PMTU is larger. Sending a packet of this size ensures that the network path supports an MTU of this size and helps mitigate amplification attacks caused by server responses toward an unverified client address.

Servers MUST reject the first plaintext packet received from a client if it its total size is less than 1280 octets, to mitigate amplification attacks.

If a QUIC endpoint determines that the PMTU between any pair of local and remote IP addresses has fallen below 1280 octets, it MUST immediately cease sending QUIC packets between those IP addresses. This may result in abrupt termination of the connection if all pairs are affected. In this case, an endpoint SHOULD send a Public Reset packet to indicate the failure. The application SHOULD attempt to use TLS over TCP instead.

A sender bundles one or more frames in a Regular QUIC packet (see Section 6).

A sender SHOULD minimize per-packet bandwidth and computational costs by bundling as many frames as possible within a QUIC packet. A sender MAY wait for a short period of time to bundle multiple frames before sending a packet that is not maximally packed, to avoid sending out large numbers of small packets. An implementation may use heuristics about expected application sending behavior to determine whether and for how long to wait. This waiting period is an implementation decision, and an implementation should be careful to delay conservatively, since any delay is likely to increase application-visible latency.

Regular QUIC packets are "containers" of frames; a packet is never retransmitted whole. How an endpoint handles the loss of the frame depends on the type of the frame. Some frames are simply retransmitted, some have their contents moved to new frames, and others are never retransmitted.

When a packet is detected as lost, the sender re-sends any frames as necessary:

- All application data sent in STREAM frames MUST be retransmitted, unless the endpoint has sent a RST_STREAM for that stream. When an endpoint sends a RST_STREAM frame, data outstanding on that stream SHOULD NOT be retransmitted, since subsequent data on this stream is expected to not be delivered by the receiver.

- ACK and PADDING frames MUST NOT be retransmitted. ACK frames are cumulative, so new frames containing updated information will be sent as described in Section 8.2.

- All other frames MUST be retransmitted.

Upon detecting losses, a sender MUST take appropriate congestion control action. The details of loss detection and congestion control are described in [QUIC-RECOVERY].

A packet MUST NOT be acknowledged until packet protection has been successfully removed and all frames contained in the packet have been processed. For STREAM frames, this means the data has been queued (but not necessarily delivered to the application). This also means that any stream state transitions triggered by STREAM or RST_STREAM frames have occurred. Once the packet has been fully processed, a receiver acknowledges receipt by sending one or more ACK frames containing the packet number of the received packet.

To avoid creating an indefinite feedback loop, an endpoint MUST NOT generate an ACK frame in response to a packet containing only ACK or PADDING frames.

Strategies and implications of the frequency of generating acknowledgments are discussed in more detail in [QUIC-RECOVERY].

9.1. Special Considerations for PMTU Discovery

Traditional ICMP-based path MTU discovery in IPv4 ([RFC1191] is potentially vulnerable to off-path attacks that successfully guess the IP/port 4-tuple and reduce the MTU to a bandwidth-inefficient value. TCP connections mitigate this risk by using the (at minimum) 8 bytes of transport header echoed in the ICMP message to validate the TCP sequence number as valid for the current connection. However, as QUIC operates over UDP, in IPv4 the echoed information could consist only of the IP and UDP headers, which usually has insufficient entropy to mitigate off-path attacks.

As a result, endpoints that implement PMTUD in IPv4 SHOULD take steps to mitigate this risk. For instance, an application could:

- Set the IPv4 Don't Fragment (DF) bit on a small proportion of packets, so that most invalid ICMP messages arrive when there are no DF packets outstanding, and can therefore be identified as spurious.
- Store additional information from the IP or UDP headers from DF packets (for example, the IP ID or UDP checksum) to further authenticate incoming Datagram Too Big messages.
- Any reduction in PMTU due to a report contained in an ICMP packet is provisional until QUIC's loss detection algorithm determines that the packet is actually lost.

10. Streams: QUIC's Data Structuring Abstraction

Streams in QUIC provide a lightweight, ordered, and bidirectional byte-stream abstraction modeled closely on HTTP/2 streams [RFC7540].

Streams can be created either by the client or the server, can concurrently send data interleaved with other streams, and can be cancelled.

Data that is received on a stream is delivered in order within that stream, but there is no particular delivery order across streams. Transmit ordering among streams is left to the implementation.

The creation and destruction of streams are expected to have minimal bandwidth and computational cost. A single STREAM frame may create, carry data for, and terminate a stream, or a stream may last the entire duration of a connection.

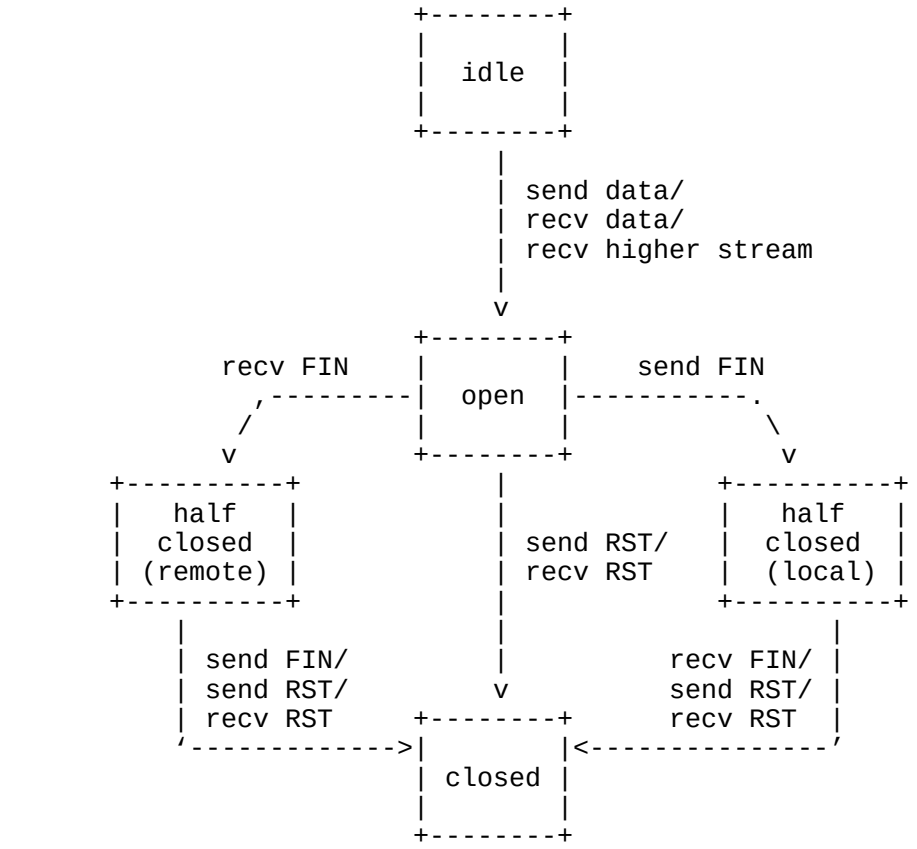
Streams are individually flow controlled, allowing an endpoint to limit memory commitment and to apply back pressure.

An alternative view of QUIC streams is as an elastic "message" abstraction, similar to the way ephemeral streams are used in SST [SST], which may be a more appealing description for some applications.

10.1. Life of a Stream

The semantics of QUIC streams is based on HTTP/2 streams, and the lifecycle of a QUIC stream therefore closely follows that of an HTTP/2 stream [RFC7540], with some differences to accommodate the possibility of out-of-order delivery due to the use of multiple

streams in QUIC. The lifecycle of a QUIC stream is shown in the following figure and described below.



send: endpoint sends this frame
recv: endpoint receives this frame

data: application data in a STREAM frame
FIN: FIN flag in a STREAM frame
RST: RST_STREAM frame

Figure 11: Lifecycle of a stream

Note that this diagram shows stream state transitions and the frames and flags that affect those transitions only. For the purpose of state transitions, the FIN flag is processed as a separate event to the frame that bears it; a STREAM frame with the FIN flag set can cause two state transitions. When the FIN flag is sent on an empty

STREAM frame, the offset in the STREAM frame MUST be one greater than the last data byte sent on this stream.

The recipient of a frame which changes stream state will have a delayed view of the state of a stream while the frame is in transit. Endpoints do not coordinate the creation of streams; they are created unilaterally by either endpoint. The negative consequences of a mismatch in states are limited to the "closed" state after sending RST_STREAM, where frames might be received for some time after closing. Endpoints can use acknowledgments to understand the peer's subjective view of stream state at any given time.

Streams have the following states:

10.1.1. idle

All streams start in the "idle" state.

The following transitions are valid from this state:

Sending or receiving a STREAM frame causes the stream to become "open". The stream identifier is selected as described in Section 10.2. The same STREAM frame can also cause a stream to immediately become "half-closed".

Receiving a STREAM frame on a peer-initiated stream (that is, a packet sent by a server on an even-numbered stream or a client packet on an odd-numbered stream) also causes all lower-numbered "idle" streams in the same direction to become "open". This could occur if a peer begins sending on streams in a different order to their creation, or it could happen if packets are lost or reordered in transit.

Receiving any frame other than STREAM or RST_STREAM on a stream in this state MUST be treated as a connection error (Section 12) of type YYYY.

10.1.2. open

A stream in the "open" state may be used by both peers to send frames of any type. In this state, a sending peer must observe the flow-control limit advertised by its receiving peer (Section 11).

From this state, either endpoint can send a frame with the FIN flag set, which causes the stream to transition into one of the "half-closed" states. An endpoint sending an FIN flag causes the stream state to become "half-closed (local)". An endpoint receiving a FIN flag causes the stream state to become "half-closed (remote)" once

all preceding data has arrived. The receiving endpoint MUST NOT consider the stream state to have changed until all data has arrived.

Either endpoint can send a RST_STREAM frame from this state, causing it to transition immediately to "closed".

10.1.3. half-closed (local)

A stream that is in the "half-closed (local)" state MUST NOT be used for sending STREAM frames; WINDOW_UPDATE and RST_STREAM MAY be sent in this state.

A stream transitions from this state to "closed" when a STREAM frame that contains a FIN flag is received and all prior data has arrived, or when either peer sends a RST_STREAM frame.

An endpoint that closes a stream MUST NOT send data beyond the final offset that it has chosen, see Section 10.1.5 for details.

An endpoint can receive any type of frame in this state. Providing flow-control credit using WINDOW_UPDATE frames is necessary to continue receiving flow-controlled frames. In this state, a receiver MAY ignore WINDOW_UPDATE frames for this stream, which might arrive for a short period after a frame bearing the FIN flag is sent.

10.1.4. half-closed (remote)

A stream that is "half-closed (remote)" is no longer being used by the peer to send any data. In this state, a sender is no longer obligated to maintain a receiver stream-level flow-control window.

A stream that is in the "half-closed (remote)" state will have a final offset for received data, see Section 10.1.5 for details.

A stream in this state can be used by the endpoint to send frames of any type. In this state, the endpoint continues to observe advertised stream-level and connection-level flow-control limits (Section 11).

A stream can transition from this state to "closed" by sending a frame that contains a FIN flag or when either peer sends a RST_STREAM frame.

10.1.5. closed

The "closed" state is the terminal state.

An endpoint will learn the final offset of the data it receives on a stream when it enters the "half-closed (remote)" or "closed" state. The final offset is carried explicitly in the RST_STREAM frame; otherwise, the final offset is the offset of the end of the data carried in STREAM frame marked with a FIN flag.

An endpoint MUST NOT send data on a stream at or beyond the final offset.

Once a final offset for a stream is known, it cannot change. If a RST_STREAM or STREAM frame causes the final offset to change for a stream, an endpoint SHOULD respond with a QUIC_STREAM_DATA_AFTER_TERMINATION error (see Section 12). A receiver SHOULD treat receipt of data at or beyond the final offset as a QUIC_STREAM_DATA_AFTER_TERMINATION error. Generating these errors is not mandatory, but only because requiring that an endpoint generate these errors also means that the endpoint needs to maintain the final offset state for closed streams, which could mean a significant state commitment.

An endpoint that receives a RST_STREAM frame (and which has not sent a FIN or a RST_STREAM) MUST immediately respond with a RST_STREAM frame, and MUST NOT send any more data on the stream. This endpoint may continue receiving frames for the stream on which a RST_STREAM is received.

If this state is reached as a result of sending a RST_STREAM frame, the peer that receives the RST_STREAM frame might have already sent - or enqueued for sending - frames on the stream that cannot be withdrawn. An endpoint MUST ignore frames that it receives on closed streams after it has sent a RST_STREAM frame. An endpoint MAY choose to limit the period over which it ignores frames and treat frames that arrive after this time as being in error.

STREAM frames received after sending RST_STREAM are counted toward the connection and stream flow-control windows. Even though these frames might be ignored, because they are sent before their sender receives the RST_STREAM, the sender will consider the frames to count against its flow-control windows.

In the absence of more specific guidance elsewhere in this document, implementations SHOULD treat the receipt of a frame that is not expressly permitted in the description of a state as a connection error (Section 12). Frames of unknown types are ignored.

(TODO: QUIC_STREAM_NO_ERROR is a special case. Write it up.)

10.2. Stream Identifiers

Streams are identified by an unsigned 32-bit integer, referred to as the StreamID. To avoid StreamID collision, clients MUST initiate streams usinge odd-numbered StreamIDs; streams initiated by the server MUST use even-numbered StreamIDs.

A StreamID of zero (0x0) is reserved and used for connection-level flow control frames (Section 11); the StreamID of zero cannot be used to establish a new stream.

StreamID 1 (0x1) is reserved for the cryptographic handshake. StreamID 1 MUST NOT be used for application data, and MUST be the first client-initiated stream.

A QUIC endpoint cannot reuse a StreamID on a given connection. Streams MUST be created in sequential order. Open streams can be used in any order. Streams that are used out of order result in lower-numbered streams in the same direction being counted as open.

All streams, including stream 1, count toward this limit. Thus, a concurrent stream limit of 0 will cause a connection to be unusable. Application protocols that use QUIC might require a certain minimum number of streams to function correctly. If a peer advertises an concurrent stream limit (concurrent_streams) that is too small for the selected application protocol to function, an endpoint MUST terminate the connection with an error of type QUIC_TOO_MANY_OPEN_STREAMS (Section 12).

10.3. Stream Concurrency

An endpoint limits the number of concurrently active incoming streams by setting the concurrent stream limit (see Section 7.3.1) in the transport parameters. The maximum concurrent streams setting is specific to each endpoint and applies only to the peer that receives the setting. That is, clients specify the maximum number of concurrent streams the server can initiate, and servers specify the maximum number of concurrent streams the client can initiate.

Streams that are in the "open" state or in either of the "half-closed" states count toward the maximum number of streams that an endpoint is permitted to open. Streams in any of these three states count toward the limit advertised in the concurrent stream limit.

A recently closed stream MUST also be considered to count toward this limit until packets containing all frames required to close the stream have been acknowledged. For a stream which closed cleanly, this means all STREAM frames have been acknowledged; for a stream

which closed abruptly, this means the RST_STREAM frame has been acknowledged.

Endpoints MUST NOT exceed the limit set by their peer. An endpoint that receives a STREAM frame that causes its advertised concurrent stream limit to be exceeded MUST treat this as a stream error of type QUIC_TOO_MANY_OPEN_STREAMS (Section 12).

10.4. Sending and Receiving Data

Once a stream is created, endpoints may use the stream to send and receive data. Each endpoint may send a series of STREAM frames encapsulating data on a stream until the stream is terminated in that direction. Streams are an ordered byte-stream abstraction, and they have no other structure within them. STREAM frame boundaries are not expected to be preserved in retransmissions from the sender or during delivery to the application at the receiver.

When new data is to be sent on a stream, a sender MUST set the encapsulating STREAM frame's offset field to the stream offset of the first byte of this new data. The first byte of data that is sent on a stream has the stream offset 0. The largest offset delivered on a stream MUST be less than 2^64. A receiver MUST ensure that received stream data is delivered to the application as an ordered byte-stream. Data received out of order MUST be buffered for later delivery, as long as it is not in violation of the receiver's flow control limits.

The cryptographic handshake stream, Stream 1, MUST NOT be subject to congestion control or connection-level flow control, but MUST be subject to stream-level flow control. An endpoint MUST NOT send data on any other stream without consulting the congestion controller and the flow controller.

Flow control is described in detail in Section 11, and congestion control is described in the companion document [QUIC-RECOVERY].

10.5. Stream Prioritization

Stream multiplexing has a significant effect on application performance if resources allocated to streams are correctly prioritized. Experience with other multiplexed protocols, such as HTTP/2 [RFC7540], shows that effective prioritization strategies have a significant positive impact on performance.

QUIC does not provide frames for exchanging priotization information. Instead it relies on receiving priority information from the application that uses QUIC. Protocols that use QUIC are able to

define any prioritization scheme that suits their application semantics. A protocol might define explicit messages for signaling priority, such as those defined in HTTP/2; it could define rules that allow an endpoint to determine priority based on context; or it could leave the determination to the application.

A QUIC implementation SHOULD provide ways in which an application can indicate the relative priority of streams. When deciding which streams to dedicate resources to, QUIC SHOULD use the information provided by the application. Failure to account for priority of streams can result in suboptimal performance.

Stream priority is most relevant when deciding which stream data will be transmitted. Often, there will be limits on what can be transmitted as a result of connection flow control or the current congestion controller state.

Giving preference to the transmission of its own management frames ensures that the protocol functions efficiently. That is, prioritizing frames other than STREAM frames ensures that loss recovery, congestion control, and flow control operate effectively.

Stream 1 MUST be prioritized over other streams prior to the completion of the cryptographic handshake. This includes the retransmission of the second flight of client handshake messages, that is, the TLS Finished and any client authentication messages.

STREAM frames that are determined to be lost SHOULD be retransmitted before sending new data, unless application priorities indicate otherwise. Retransmitting lost STREAM frames can fill in gaps, which allows the peer to consume already received data and free up flow control window.

11. Flow Control

It is necessary to limit the amount of data that a sender may have outstanding at any time, so as to prevent a fast sender from overwhelming a slow receiver, or to prevent a malicious sender from consuming significant resources at a receiver. This section describes QUIC's flow-control mechanisms.

QUIC employs a credit-based flow-control scheme similar to HTTP/2's flow control [RFC7540]. A receiver advertises the number of octets it is prepared to receive on a given stream and for the entire connection. This leads to two levels of flow control in QUIC: (i) Connection flow control, which prevents senders from exceeding a receiver's buffer capacity for the connection, and (ii) Stream flow

control, which prevents a single stream from consuming the entire receive buffer for a connection.

A receiver sends WINDOW_UPDATE frames to the sender to advertise additional credit by sending the absolute byte offset in the stream or in the connection which it is willing to receive.

The initial flow control credit is 65536 bytes for both the stream and connection flow controllers.

A receiver MAY advertise a larger offset at any point in the connection by sending a WINDOW_UPDATE frame. A receiver MUST NOT renege on an advertisement; that is, once a receiver advertises an offset via a WINDOW_UPDATE frame, it MUST NOT subsequently advertise a smaller offset. A sender may receive WINDOW_UPDATE frames out of order; a sender MUST therefore ignore any WINDOW_UPDATE that does not move the window forward.

A receiver MUST close the connection with a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA error (Section 12) if the peer violates the advertised stream or connection flow control windows.

A sender MUST send BLOCKED frames to indicate it has data to write but is blocked by lack of connection or stream flow control credit. BLOCKED frames are expected to be sent infrequently in common cases, but they are considered useful for debugging and monitoring purposes.

A receiver advertises credit for a stream by sending a WINDOW_UPDATE frame with the StreamID set appropriately. A receiver may use the current offset of data consumed to determine the flow control offset to be advertised. A receiver MAY send copies of a WINDOW_UPDATE frame in multiple packets in order to make sure that the sender receives it before running out of flow control credit, even if one of the packets is lost.

Connection flow control is a limit to the total bytes of stream data sent in STREAM frames on all streams contributing to connection flow control. A receiver advertises credit for a connection by sending a WINDOW_UPDATE frame with the StreamID set to zero (0x00). A receiver maintains a cumulative sum of bytes received on all streams contributing to connection-level flow control, to check for flow control violations. A receiver may maintain a cumulative sum of bytes consumed on all contributing streams to determine the connection-level flow control offset to be advertised.

11.1. Edge Cases and Other Considerations

There are some edge cases which must be considered when dealing with stream and connection level flow control. Given enough time, both endpoints must agree on flow control state. If one end believes it can send more than the other end is willing to receive, the connection will be torn down when too much data arrives. Conversely if a sender believes it is blocked, while endpoint B expects more data can be received, then the connection can be in a deadlock, with the sender waiting for a WINDOW_UPDATE which will never come.

11.1.1. Mid-stream RST_STREAM

On receipt of a RST_STREAM frame, an endpoint will tear down state for the matching stream and ignore further data arriving on that stream. This could result in the endpoints getting out of sync, since the RST_STREAM frame may have arrived out of order and there may be further bytes in flight. The data sender would have counted the data against its connection level flow control budget, but a receiver that has not received these bytes would not know to include them as well. The receiver must learn the number of bytes that were sent on the stream to make the same adjustment in its connection flow controller.

To avoid this de-synchronization, a RST_STREAM sender MUST include the final byte offset sent on the stream in the RST_STREAM frame. On receiving a RST_STREAM frame, a receiver definitively knows how many bytes were sent on that stream before the RST_STREAM frame, and the receiver MUST use the final offset to account for all bytes sent on the stream in its connection level flow controller.

11.1.2. Response to a RST_STREAM

Since streams are bidirectional, a sender of a RST_STREAM needs to know how many bytes the peer has sent on the stream. If an endpoint receives a RST_STREAM frame and has sent neither a FIN nor a RST_STREAM, it MUST send a RST_STREAM in response, bearing the offset of the last byte sent on this stream as the final offset.

11.1.3. Offset Increment

This document leaves when and how many bytes to advertise in a WINDOW_UPDATE to the implementation, but offers a few considerations. WINDOW_UPDATE frames constitute overhead, and therefore, sending a WINDOW_UPDATE with small offset increments is undesirable. At the same time, sending WINDOW_UPDATES with large offset increments requires the sender to commit to that amount of buffer.

Implementations must find the correct tradeoff between these sides to determine how large an offset increment to send in a WINDOW_UPDATE.

A receiver MAY use an autotuning mechanism to tune the size of the offset increment to advertise based on a roundtrip time estimate and the rate at which the receiving application consumes data, similar to common TCP implementations.

11.1.4. BLOCKED frames

If a sender does not receive a WINDOW_UPDATE frame when it has run out of flow control credit, the sender will be blocked and MUST send a BLOCKED frame. A BLOCKED frame is expected to be useful for debugging at the receiver. A receiver SHOULD NOT wait for a BLOCKED frame before sending a WINDOW_UPDATE, since doing so will cause at least one roundtrip of quiescence. For smooth operation of the congestion controller, it is generally considered best to not let the sender go into quiescence if avoidable. To avoid blocking a sender, and to reasonably account for the possibiity of loss, a receiver should send a WINDOW_UPDATE frame at least two roundtrips before it expects the sender to get blocked.

12. Error Handling

An endpoint that detects an error SHOULD signal the existence of that error to its peer. Errors can affect an entire connection (see Section 12.1), or a single stream (see Section 12.2).

The most appropriate error code (Section 12.3) SHOULD be included in the frame that signals the error. Where this specification identifies error conditions, it also identifies the error code that is used.

Public Reset is not suitable for any error that can be signaled with a CONNECTION_CLOSE or RST_STREAM frame. Public Reset MUST NOT be sent by an endpoint that has the state necessary to send a frame on the connection.

12.1. Connection Errors

Errors that result in the connection being unusable, such as an obvious violation of protocol semantics or corruption of state that affects an entire connection, MUST be signaled using a CONNECTION_CLOSE frame (Section 8.8). An endpoint MAY close the connection in this manner, even if the error only affects a single stream.

A CONNECTION_CLOSE frame could be sent in a packet that is lost. An endpoint SHOULD be prepared to retransmit a packet containing a CONNECTION_CLOSE frame if it receives more packets on a terminated connection. Limiting the number of retransmissions and the time over which this final packet is sent limits the effort expended on terminated connections.

An endpoint that chooses not to retransmit packets containing CONNECTION_CLOSE risks a peer missing the first such packet. The only mechanism available to an endpoint that continues to receive data for a terminated connection is to send a Public Reset packet.

12.2. Stream Errors

If the error affects a single stream, but otherwise leaves the connection in a recoverable state, the endpoint can sent a RST_STREAM frame (Section 8.5) with an appropriate error code to terminate just the affected stream.

Stream 1 is critical to the functioning of the entire connection. If stream 1 is closed with either a RST_STREAM or STREAM frame bearing the FIN flag, an endpoint MUST generate a connection error of type QUIC_CLOSED_CRITICAL_STREAM.

Some application protocols make other streams critical to that protocol. An application protocol does not need to inform the transport that a stream is critical; it can instead generate appropriate errors in response to being notified that the critical stream is closed.

An endpoint MAY send a RST_STREAM frame in the same packet as a CONNECTION_CLOSE frame.

12.3. Error Codes

Error codes are 32 bits long, with the first two bits indicating the source of the error code:

0x00000000-0x3FFFFFFF: Application-specific error codes. Defined by each application-layer protocol.

0x40000000-0x7FFFFFFF: Reserved for host-local error codes. These codes MUST NOT be sent to a peer, but MAY be used in API return codes and logs.

0x80000000-0xBFFFFFFF: QUIC transport error codes, including packet protection errors. Applicable to all uses of QUIC.

0xC0000000-0xFFFFFFFF: Cryptographic error codes. Defined by the cryptographic handshake protocol in use.

This section lists the defined QUIC transport error codes that may be used in a CONNECTION_CLOSE or RST_STREAM frame. Error codes share a common code space. Some error codes apply only to either streams or the entire connection and have no defined semantics in the other context.

QUIC_INTERNAL_ERROR (0x80000001): Connection has reached an invalid state.

QUIC_STREAM_DATA_AFTER_TERMINATION (0x80000002): There were data frames after the a fin or reset.

QUIC_INVALID_PACKET_HEADER (0x80000003): Control frame is malformed.

QUIC_INVALID_FRAME_DATA (0x80000004): Frame data is malformed.

QUIC_MULTIPLE_TERMINATION_OFFSETS (0x80000005): Multiple final offset values were received on the same stream

QUIC_STREAM_CANCELLED (0x80000006): The stream was cancelled

QUIC_CLOSED_CRITICAL_STREAM (0x80000007): A stream that is critical to the protocol was closed.

QUIC_MISSING_PAYLOAD (0x80000030): The packet contained no payload.

QUIC_INVALID_STREAM_DATA (0x8000002E): STREAM frame data is malformed.

QUIC_UNENCRYPTED_STREAM_DATA (0x8000003D): Received STREAM frame data is not encrypted.

QUIC_MAYBE_CORRUPTED_MEMORY (0x80000059): Received a frame which is likely the result of memory corruption.

QUIC_INVALID_RST_STREAM_DATA (0x80000006): RST_STREAM frame data is malformed.

QUIC_INVALID_CONNECTION_CLOSE_DATA (0x80000007): CONNECTION_CLOSE frame data is malformed.

QUIC_INVALID_GOAWAY_DATA (0x80000008): GOAWAY frame data is malformed.

QUIC_INVALID_WINDOW_UPDATE_DATA (0x80000039): WINDOW_UPDATE frame data is malformed.

QUIC_INVALID_BLOCKED_DATA (0x8000003A): BLOCKED frame data is malformed.

QUIC_INVALID_PATH_CLOSE_DATA (0x8000004E): PATH_CLOSE frame data is malformed.

QUIC_INVALID_ACK_DATA (0x80000009): ACK frame data is malformed.

QUIC_INVALID_VERSION_NEGOTIATION_PACKET (0x8000000A): Version negotiation packet is malformed.

QUIC_INVALID_PUBLIC_RST_PACKET (0x8000000b): Public RST packet is malformed.

QUIC_DECRYPTION_FAILURE (0x8000000c): There was an error decrypting.

QUIC_ENCRYPTION_FAILURE (0x8000000d): There was an error encrypting.

QUIC_PACKET_TOO_LARGE (0x8000000e): The packet exceeded kMaxPacketSize.

QUIC_PEER_GOING_AWAY (0x80000010): The peer is going away. May be a client or server.

QUIC_INVALID_STREAM_ID (0x80000011): A stream ID was invalid.

QUIC_INVALID_PRIORITY (0x80000031): A priority was invalid.

QUIC_TOO_MANY_OPEN_STREAMS (0x80000012): Too many streams already open.

QUIC_TOO_MANY_AVAILABLE_STREAMS (0x8000004c): The peer created too many available streams.

QUIC_PUBLIC_RESET (0x80000013): Received public reset for this connection.

QUIC_INVALID_VERSION (0x80000014): Invalid protocol version.

QUIC_INVALID_HEADER_ID (0x80000016): The Header ID for a stream was too far from the previous.

QUIC_INVALID_NEGOTIATED_VALUE (0x80000017): Negotiable parameter received during handshake had invalid value.

QUIC_DECOMPRESSION_FAILURE (0x80000018): There was an error decompressing data.

QUIC_NETWORK_IDLE_TIMEOUT (0x80000019): The connection timed out due to no network activity.

QUIC_HANDSHAKE_TIMEOUT (0x80000043): The connection timed out waiting for the handshake to complete.

QUIC_ERROR_MIGRATING_ADDRESS (0x8000001a): There was an error encountered migrating addresses.

QUIC_ERROR_MIGRATING_PORT (0x80000056): There was an error encountered migrating port only.

QUIC_EMPTY_STREAM_FRAME_NO_FIN (0x80000032): We received a STREAM_FRAME with no data and no fin flag set.

QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA (0x8000003b): The peer received too much data, violating flow control.

QUIC_FLOW_CONTROL_SENT_TOO_MUCH_DATA (0x8000003f): The peer sent too much data, violating flow control.

QUIC_FLOW_CONTROL_INVALID_WINDOW (0x80000040): The peer received an invalid flow control window.

QUIC_CONNECTION_IP_POOLED (0x8000003e): The connection has been IP pooled into an existing connection.

QUIC_TOO_MANY_OUTSTANDING_SENT_PACKETS (0x80000044): The connection has too many outstanding sent packets.

QUIC_TOO_MANY_OUTSTANDING_RECEIVED_PACKETS (0x80000045): The connection has too many outstanding received packets.

QUIC_CONNECTION_CANCELLED (0x80000046): The QUIC connection has been cancelled.

QUIC_BAD_PACKET_LOSS_RATE (0x80000047): Disabled QUIC because of high packet loss rate.

QUIC_PUBLIC_RESETS_POST_HANDSHAKE (0x80000049): Disabled QUIC because of too many PUBLIC_RESETs post handshake.

QUIC_TIMEOUTS_WITH_OPEN_STREAMS (0x8000004a): Disabled QUIC because of too many timeouts with streams open.

QUIC_TOO_MANY_RTOS (0x80000055): QUIC timed out after too many RTOs.

QUIC_ENCRYPTION_LEVEL_INCORRECT (0x8000002c): A packet was received with the wrong encryption level (i.e. it should have been encrypted but was not.)

QUIC_VERSION_NEGOTIATION_MISMATCH (0x80000037): This connection involved a version negotiation which appears to have been tampered with.

QUIC_IP_ADDRESS_CHANGED (0x80000050): IP address changed causing connection close.

QUIC_ADDRESS_VALIDATION_FAILURE (0x80000051): Client address validation failed.

QUIC_TOO_MANY_FRAME_GAPS (0x8000005d): Stream frames arrived too discontiguously so that stream sequencer buffer maintains too many gaps.

QUIC_TOO_MANY_SESSIONS_ON_SERVER (0x80000060): Connection closed because server hit max number of sessions allowed.

13. Security and Privacy Considerations

13.1. Spoofed ACK Attack

An attacker receives an STK from the server and then releases the IP address on which it received the STK. The attacker may, in the future, spoof this same address (which now presumably addresses a different endpoint), and initiate a 0-RTT connection with a server on the victim's behalf. The attacker then spoofs ACK frames to the server which cause the server to potentially drown the victim in data.

There are two possible mitigations to this attack. The simplest one is that a server can unilaterally create a gap in packet-number space. In the non-attack scenario, the client will send an ACK frame with the larger value for largest acknowledged. In the attack scenario, the attacker could acknowledge a packet in the gap. If the server sees an acknowledgment for a packet that was never sent, the connection can be aborted.

The second mitigation is that the server can require that acknowledgments for sent packets match the encryption level of the sent packet. This mitigation is useful if the connection has an ephemeral forward-secure key that is generated and used for every new connection. If a packet sent is encrypted with a forward-secure key,

then any acknowledgments that are received for them MUST also be forward-secure encrypted. Since the attacker will not have the forward secure key, the attacker will not be able to generate forward-secure encrypted packets with ACK frames.

14. IANA Considerations

14.1. QUIC Transport Parameter Registry

IANA [SHALL add/has added] a registry for "QUIC Transport Parameters" under a "QUIC Protocol" heading.

The "QUIC Transport Parameters" registry governs a 16-bit space. This space is split into two spaces that are governed by different policies. Values with the first byte in the range 0x00 to 0xfe (in hexadecimal) are assigned via the Specification Required policy [RFC5226]. Values with the first byte 0xff are reserved for Private Use [RFC5226].

Registrations MUST include the following fields:

Value: The numeric value of the assignment (registrations will be between 0x0000 and 0xfeff).

Parameter Name: A short mnemonic for the parameter.

Specification: A reference to a publicly available specification for the value.

The nominated expert(s) verify that a specification exists and is readily accessible. The expert(s) are encouraged to be biased towards approving registrations unless they are abusive, frivolous, or actively harmful (not merely aesthetically displeasing, or architecturally dubious).

The initial contents of this registry are shown in Table 4.

Value	Parameter Name	Specification
0x0000	stream_fc_offset	Section 7.3.1
0x0001	connection_fc_offset	Section 7.3.1
0x0002	concurrent_streams	Section 7.3.1
0x0003	idle_timeout	Section 7.3.1
0x0004	truncate_connection_id	Section 7.3.1

Table 4: Initial QUIC Transport Parameters Entries

15. References

15.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-19 (work in progress), March 2017.

[QUIC-RECOVERY]
Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control".

[QUIC-TLS]
Thomson, M., Ed. and S. Turner, Ed., "Using Transport Layer Security (TLS) to Secure QUIC".

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <http://www.rfc-editor.org/info/rfc1191>.

[RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, DOI 10.17487/RFC1981, August 1996, <http://www.rfc-editor.org/info/rfc1981>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <http://www.rfc-editor.org/info/rfc4821>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

15.2. Informative References

[EARLY-DESIGN] Roskind, J., "QUIC: Multiplexed Transport Over UDP", December 2013, <https://goo.gl/dMVtFi>.

[RFC2360] Scott, G., "Guide for Internet Standards Writers", BCP 22, RFC 2360, DOI 10.17487/RFC2360, June 1998, <http://www.rfc-editor.org/info/rfc2360>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <http://www.rfc-editor.org/info/rfc4086>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <http://www.rfc-editor.org/info/rfc6824>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <http://www.rfc-editor.org/info/rfc7301>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <http://www.rfc-editor.org/info/rfc7540>.

[SST] Ford, B., "Structured Streams: A New Transport Abstraction", DOI 10.1145/1282427.1282421, ACM SIGCOMM Computer Communication Review Volume 37 Issue 4, October 2007.

15.3. URIs

[1] https://github.com/quicwg/base-drafts/wiki/QUIC-Versions

Appendix A. Contributors

The original authors of this specification were Ryan Hamilton, Jana Iyengar, Ian Swett, and Alyssa Wilk.

The original design and rationale behind this protocol draw significantly from work by Jim Roskind [EARLY-DESIGN]. In alphabetical order, the contributors to the pre-IETF QUIC project at Google are: Britt Cyr, Jeremy Dorfman, Ryan Hamilton, Jana Iyengar, Fedor Kouranov, Charles Krasic, Jo Kulik, Adam Langley, Jim Roskind, Robbie Shade, Satyam Shekhar, Cherie Shi, Ian Swett, Raman Tenneti, Victor Vasiliev, Antonio Vicente, Patrik Westin, Alyssa Wilk, Dale Worley, Fan Yang, Dan Zhang, Daniel Ziegler.

Appendix B. Acknowledgments

Special thanks are due to the following for helping shape pre-IETF QUIC and its deployment: Chris Bentzel, Misha Efimov, Roberto Peon, Alistair Riddoch, Siddharth Vijayakrishnan, and Assar Westerlund.

This document has benefited immensely from various private discussions and public ones on the quic@ietf.org and proto-quic@chromium.org mailing lists. Our thanks to all.

Appendix C. Change Log

> *RFC Editor's Note:* Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

C.1. Since draft-ietf-quic-transport-01:

- Defined short and long packet headers (#40, #148, #361)
- Defined a versioning scheme and stable fields (#51, #361)
- Define reserved version values for "greasing" negotiation (#112, #278)
- The initial packet number is randomized (#35, #283)
- Narrow the packet number encoding range requirement (#67, #286, #299, #323, #356)

- Defined client address validation (#52, #118, #120, #275)
- Define transport parameters as a TLS extension (#122)
- SCUP and COPT parameters are no longer valid (#116, #117)
- Transport parameters for 0-RTT are either remembered from before, or assume default values (#126)
- The server chooses connection IDs in its final flight (#119, #349, #361)
- The server echoes the Connection ID and packet number fields when sending a Version Negotiation packet (#133, #295, #244)
- Definied a minimum packet size for the initial handshake packet from the client (#69, #136, #139, #164)
- Path MTU Discovery (#64, #106)
- The initial handshake packet from the client needs to fit in a single packet (#338)
- Forbid acknowledgment of packets containing only ACK and PADDING (#291)
- Require that frames are processed when packets are acknowledged (#381, #341)
- Removed the STOP_WAITING frame (#66)
- Don’t require retransmission of old timestamps for lost ACK frames (#308)
- Clarified that frames are not retransmitted, but the information in them can be (#157, #298)
- Error handling definitions (#335)
- Split error codes into four sections (#74)
- Forbid the use of Public Reset where CONNECTION_CLOSE is possible (#289)
- Define packet protection rules (#336)

- Require that stream be entirely delivered or reset, including acknowledgment of all STREAM frames or the RST_STREAM, before it closes (#381)
- Remove stream reservation from state machine (#174, #280)
- Only stream 0 does not contributing to connection-level flow control (#204)
- Stream 1 counts towards the maximum concurrent stream limit (#201, #282)
- Remove connection-level flow control exclusion for some streams (except 1) (#246)
- RST_STREAM affects connection-level flow control (#162, #163)
- Flow control accounting uses the maximum data offset on each stream, rather than bytes received (#378)
- Moved length-determining fields to the start of STREAM and ACK (#168, #277)
- Added the ability to pad between frames (#158, #276)
- Remove error code and reason phrase from GOAWAY (#352, #355)
- GOAWAY includes a final stream number for both directions (#347)
- Error codes for RST_STREAM and CONNECTION_CLOSE are now at a consistent offset (#249)
- Defined priority as the responsibility of the application protocol (#104, #303)

C.2. Since draft-ietf-quic-transport-00:

- Replaced DIVERSIFICATION_NONCE flag with KEY_PHASE flag
- Defined versioning
- Reworked description of packet and frame layout
- Error code space is divided into regions for each component
- Use big endian for all numeric values

C.3. Since draft-hamilton-quic-transport-protocol-01:

- Adopted as base for draft-ietf-quic-tls.
- Updated authors/editors list.
- Added IANA Considerations section.
- Moved Contributors and Acknowledgments to appendices.

Authors' Addresses

Jana Iyengar (editor)
Google

Email: jri@google.com

Martin Thomson (editor)
Mozilla

Email: martin.thomson@gmail.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: October 19, 2018

J. Iyengar, Ed.
Fastly
M. Thomson, Ed.
Mozilla

April 17, 2018

QUIC: A UDP-Based Multiplexed and Secure Transport
draft-ietf-quic-transport-11

Abstract

This document defines the core of the QUIC transport protocol. This document describes connection establishment, packet format, multiplexing and reliability. Accompanying documents describe the cryptographic handshake and loss detection.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic [1].

Working Group information can be found at https://github.com/quicwg [2]; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/-transport [3].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 19, 2018.

Table of Contents

1. Introduction

QUIC is a multiplexed and secure transport protocol that runs on top of UDP. QUIC aims to provide a flexible set of features that allow it to be a general-purpose secure transport for multiple applications.

- Version negotiation
- Low-latency connection establishment
- Authenticated and encrypted header and payload
- Stream multiplexing
- Stream and connection-level flow control
- Connection migration and resilience to NAT rebinding

QUIC implements techniques learned from experience with TCP, SCTP and other transport protocols. QUIC uses UDP as substrate so as to not require changes to legacy client operating systems and middleboxes to be deployable. QUIC authenticates all of its headers and encrypts most of the data it exchanges, including its signaling. This allows the protocol to evolve without incurring a dependency on upgrades to middleboxes. This document describes the core QUIC protocol, including the conceptual design, wire format, and mechanisms of the QUIC protocol for connection establishment, stream multiplexing, stream and connection-level flow control, connection migration, and data reliability.

Accompanying documents describe QUIC's loss detection and congestion control [QUIC-RECOVERY], and the use of TLS 1.3 for key negotiation [QUIC-TLS].

QUIC version 1 conforms to the protocol invariants in [QUIC-INVARIANTS].

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Definitions of terms that are used in this document:

Client: The endpoint initiating a QUIC connection.

Server: The endpoint accepting incoming QUIC connections.

Endpoint: The client or server end of a connection.

Stream: A logical, bi-directional channel of ordered bytes within a QUIC connection.

Connection: A conversation between two QUIC endpoints with a single encryption context that multiplexes streams within it.

Connection ID: An opaque identifier that is used to identify a QUIC connection at an endpoint. Each endpoint sets a value that its peer includes in packets.

QUIC packet: A well-formed UDP payload that can be parsed by a QUIC receiver.

QUIC is a name, not an acronym.

2.1. Notational Conventions

Packet and frame diagrams use the format described in Section 3.1 of [RFC2360], with the following additional conventions:

[x] Indicates that x is optional

x (A) Indicates that x is A bits long

x (A/B/C) ... Indicates that x is one of A, B, or C bits long

x (i) ... Indicates that x uses the variable-length encoding in Section 7.1

x (*) ... Indicates that x is variable-length

3. Versions

QUIC versions are identified using a 32-bit unsigned number.

The version 0x00000000 is reserved to represent version negotiation. This version of the specification is identified by the number 0x00000001.

Other versions of QUIC might have different properties to this version. The properties of QUIC that are guaranteed to be consistent across all versions of the protocol are described in [QUIC-INVARIANTS].

Version 0x00000001 of QUIC uses TLS as a cryptographic handshake protocol, as described in [QUIC-TLS].

Versions with the most significant 16 bits of the version number cleared are reserved for use in future IETF consensus documents.

Versions that follow the pattern 0x?a?a?a?a are reserved for use in forcing version negotiation to be exercised. That is, any version number where the low four bits of all octets is 1010 (in binary). A client or server MAY advertise support for any of these reserved versions.

Reserved version numbers will probably never represent a real protocol; a client MAY use one of these version numbers with the expectation that the server will initiate version negotiation; a server MAY advertise support for one of these versions and can expect that clients ignore the value.

[[RFC editor: please remove the remainder of this section before publication.]]

The version number for the final version of this specification (0x00000001), is reserved for the version of the protocol that is published as an RFC.

Version numbers used to identify IETF drafts are created by adding the draft number to 0xff000000. For example, draft-ietf-quic-transport-13 would be identified as 0xff00000D.

Implementors are encouraged to register version numbers of QUIC that they are using for private experimentation on the github wiki [4].

4. Packet Types and Formats

We first describe QUIC's packet types and their formats, since some are referenced in subsequent mechanisms.

All numeric values are encoded in network byte order (that is, big-endian) and all field sizes are in bits. When discussing individual bits of fields, the least significant bit is referred to as bit 0. Hexadecimal notation is used for describing the value of fields.

Any QUIC packet has either a long or a short header, as indicated by the Header Form bit. Long headers are expected to be used early in the connection before version negotiation and establishment of 1-RTT keys. Short headers are minimal version-specific headers, which are used after version negotiation and 1-RTT keys are established.

4.1. Long Header

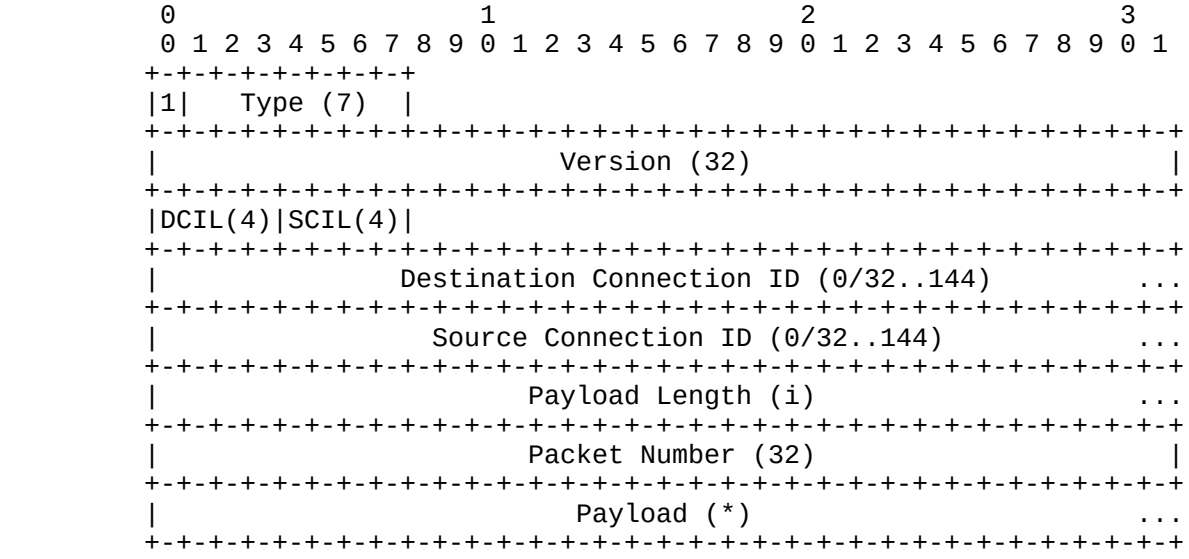


Figure 1: Long Header Format

Long headers are used for packets that are sent prior to the completion of version negotiation and establishment of 1-RTT keys. Once both conditions are met, a sender switches to sending packets using the short header (Section 4.2). The long form allows for special packets - such as the Version Negotiation packet - to be represented in this uniform fixed-length packet format. A long header contains the following fields:

Header Form: The most significant bit (0x80) of octet 0 (the first octet) is set to 1 for long headers.

Long Packet Type: The remaining seven bits of octet 0 contain the packet type. This field can indicate one of 128 packet types. The types specified for this version are listed in Table 1.

Version: The QUIC Version is a 32-bit field that follows the Type. This field indicates which version of QUIC is in use and determines how the rest of the protocol fields are interpreted.

DCIL and SCIL: Octet 1 contains the lengths of the two connection ID fields that follow it. These lengths are encoded as two 4-bit unsigned integers. The Destination Connection ID Length (DCIL) field occupies the 4 high bits of the octet and the Source Connection ID Length (SCIL) field occupies the 4 low bits of the octet. An encoded length of 0 indicates that the connection ID is also 0 octets in length. Non-zero encoded lengths are increased by 3 to get the full length of the connection ID, producing a length between 4 and 18 octets inclusive. For example, an octet with the value 0x50 describes an 8-octet Destination Connection ID and a zero-length Source Connection ID.

Destination Connection ID: The Destination Connection ID field follows the connection ID lengths and is either 0 octets in length or between 4 and 18 octets. Section 4.7 describes the use of this field in more detail.

Source Connection ID: The Source Connection ID field follows the Destination Connection ID and is either 0 octets in length or between 4 and 18 octets. Section 4.7 describes the use of this field in more detail.

Payload Length: The length of the Payload field in octets, encoded as a variable-length integer (Section 7.1).

Packet Number: The Packet Number is a 32-bit field that follows the two connection IDs. Section 4.8 describes the use of packet numbers.

Payload: The payload of the packet.

The following packet types are defined:

Type	Name	Section
0x7F	Initial	Section 4.4.1
0x7E	Retry	Section 4.4.2
0x7D	Handshake	Section 4.4.3
0x7C	0-RTT Protected	Section 4.5

Table 1: Long Header Packet Types

The header form, type, connection ID lengths octet, destination and source connection IDs, and version fields of a long header packet are version-independent. The packet number and values for packet types defined in Table 1 are version-specific. See [QUIC-INVARIANTS] for details on how packets from different versions of QUIC are interpreted.

The interpretation of the fields and the payload are specific to a version and packet type. Type-specific semantics for this version are described in the following sections.

End of the Payload field (which is also the end of the long header packet) is determined by the value of the Payload Length field. Senders can coalesce multiple long header packets into one UDP datagram. See Section 4.6 for more details.

4.2. Short Header

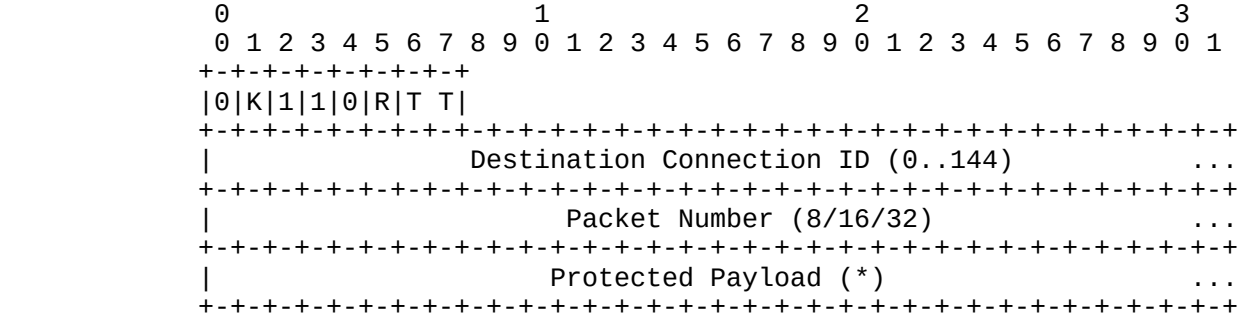


Figure 2: Short Header Format

The short header can be used after the version and 1-RTT keys are negotiated. This header form has the following fields:

Header Form: The most significant bit (0x80) of octet 0 is set to 0 for the short header.

Key Phase Bit: The second bit (0x40) of octet 0 indicates the key phase, which allows a recipient of a packet to identify the packet protection keys that are used to protect the packet. See [QUIC-TLS] for details.

[[Editor's Note: this section should be removed and the bit definitions changed before this draft goes to the IESG.]]

Third Bit: The third bit (0x20) of octet 0 is set to 1.

[[Editor's Note: this section should be removed and the bit definitions changed before this draft goes to the IESG.]]

Fourth Bit: The fourth bit (0x10) of octet 0 is set to 1.

[[Editor's Note: this section should be removed and the bit definitions changed before this draft goes to the IESG.]]

Google QUIC Demultipexing Bit: The fifth bit (0x8) of octet 0 is set to 0. This allows implementations of Google QUIC to distinguish Google QUIC packets from short header packets sent by a client because Google QUIC servers expect the connection ID to always be present. The special interpretation of this bit SHOULD be removed from this specification when Google QUIC has finished transitioning to the new header format.

Reserved: The sixth bit (0x4) of octet 0 is reserved for experimentation.

Short Packet Type: The remaining 2 bits of octet 0 include one of 4 packet types. Table 2 lists the types that are defined for short packets.

Destination Connection ID: The Destination Connection ID is a connection ID that is chosen by the intended recipient of the packet. See Section 4.7 for more details.

Packet Number: The length of the packet number field depends on the packet type. This field can be 1, 2 or 4 octets long depending on the short packet type.

Protected Payload: Packets with a short header always include a 1-RTT protected payload.

The packet type in a short header currently determines only the size of the packet number field. Additional types can be used to signal the presence of other fields.

Type	Packet Number Size
0x0	1 octet
0x1	2 octets
0x2	4 octets

Table 2: Short Header Packet Types

The header form and connection ID field of a short header packet are version-independent. The remaining fields are specific to the selected QUIC version. See [QUIC-INVARIANTS] for details on how packets from different versions of QUIC are interpreted.

4.3. Version Negotiation Packet

A Version Negotiation packet is inherently not version-specific, and does not use the long packet header (see Section 4.1. Upon receipt by a client, it will appear to be a packet using the long header, but will be identified as a Version Negotiation packet based on the Version field having a value of 0.

The Version Negotiation packet is a response to a client packet that contains a version that is not supported by the server, and is only sent by servers.

The layout of a Version Negotiation packet is:

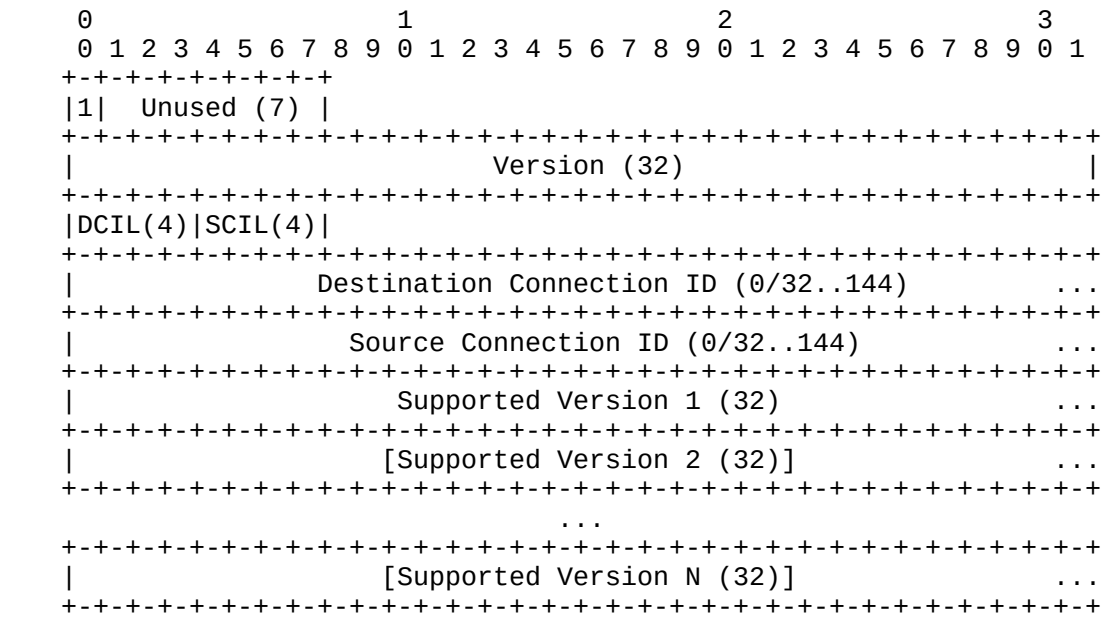


Figure 3: Version Negotiation Packet

The value in the Unused field is selected randomly by the server.

The Version field of a Version Negotiation packet MUST be set to 0x00000000.

The server MUST include the value from the Source Connection ID field of the packet it receives in the Destination Connection ID field. The value for Source Connection ID MUST be copied from the Destination Connection ID of the received packet, which is initially randomly selected by a client. Echoing both connection IDs gives clients some assurance that the server received the packet and that the Version Negotiation packet was not generated by an off-path attacker.

The remainder of the Version Negotiation packet is a list of 32-bit versions which the server supports.

A Version Negotiation packet cannot be explicitly acknowledged in an ACK frame by a client. Receiving another Initial packet implicitly acknowledges a Version Negotiation packet.

The Version Negotiation packet does not include the Packet Number and Length fields present in other packets that use the long header form.

Consequently, a Version Negotiation packet consumes an entire UDP datagram.

See Section 6.2 for a description of the version negotiation process.

4.4. Cryptographic Handshake Packets

Once version negotiation is complete, the cryptographic handshake is used to agree on cryptographic keys. The cryptographic handshake is carried in Initial (Section 4.4.1), Retry (Section 4.4.2) and Handshake (Section 4.4.3) packets.

All these packets use the long header and contain the current QUIC version in the version field.

In order to prevent tampering by version-unaware middleboxes, handshake packets are protected with a connection- and version-specific key, as described in [QUIC-TLS]. This protection does not provide confidentiality or integrity against on-path attackers, but provides some level of protection against off-path attackers.

4.4.1. Initial Packet

The Initial packet uses long headers with a type value of 0x7F. It carries the first cryptographic handshake message sent by the client.

If the client has not previously received a Retry packet from the server, it populates the Destination Connection ID field with a randomly selected value. This MUST be at least 8 octets in length. Until a packet is received from the server, the client MUST use the same random value unless it also changes the Source Connection ID (which effectively starts a new connection attempt). The randomized Destination Connection ID is used to determine packet protection keys, but is not included in server packets.

If the client received a Retry packet and is sending a second Initial packet, then it sets the Destination Connection ID to the value from the Source Connection ID in the Retry packet. Changing Destination Connection ID also results in a change to the keys used to protect the Initial packet.

The client populates the Source Connection ID field with a value of its choosing and sets the low bits of the ConnID Len field to match.

The first Initial packet that is sent by a client contains a randomized packet number. All subsequent packets contain a packet number that is incremented by one, see (Section 4.8).

The payload of an Initial packet conveys a STREAM frame (or frames) for stream 0 containing a cryptographic handshake message. The stream in this packet always starts at an offset of 0 (see Section 6.5) and the complete cryptographic handshake message MUST fit in a single packet (see Section 6.3).

The payload of a UDP datagram carrying the Initial packet MUST be expanded to at least 1200 octets (see Section 8), by adding PADDING frames to the Initial packet and/or by combining the Initial packet with a 0-RTT packet (see Section 4.6).

The client uses the Initial packet type for any packet that contains an initial cryptographic handshake message. This includes all cases where a new packet containing the initial cryptographic message needs to be created, this includes the packets sent after receiving a Version Negotiation (Section 4.3) or Retry packet (Section 4.4.2).

4.4.2. Retry Packet

A Retry packet uses long headers with a type value of 0x7E. It carries cryptographic handshake messages and acknowledgments. It is used by a server that wishes to perform a stateless retry (see Section 6.5).

The server populates the Destination Connection ID with the connection ID that the client included in the Source Connection ID of the Initial packet. This might be a zero-length value.

The server includes a connection ID of its choice in the Source Connection ID field. The client MUST use this connection ID in the Destination Connection ID of subsequent packets that it sends.

The packet number field echoes the packet number field from the triggering client packet.

A Retry packet is never explicitly acknowledged in an ACK frame by a client. Receiving another Initial packet implicitly acknowledges a Retry packet.

After receiving a Retry packet, the client uses a new Initial packet containing the next cryptographic handshake message. The client retains the state of its cryptographic handshake, but discards all transport state. The Initial packet that is generated in response to a Retry packet includes STREAM frames on stream 0 that start again at an offset of 0.

Continuing the cryptographic handshake is necessary to ensure that an attacker cannot force a downgrade of any cryptographic parameters.

In addition to continuing the cryptographic handshake, the client MUST remember the results of any version negotiation that occurred (see Section 6.2). The client MAY also retain any observed RTT or congestion state that it has accumulated for the flow, but other transport state MUST be discarded.

The payload of the Retry packet contains at least two frames. It MUST include a STREAM frame on stream 0 with offset 0 containing the server's cryptographic stateless retry material. It MUST also include an ACK frame to acknowledge the client's Initial packet. It MAY additionally include PADDING frames. The next STREAM frame sent by the server will also start at stream offset 0.

4.4.3. Handshake Packet

A Handshake packet uses long headers with a type value of 0x7D. It is used to carry acknowledgments and cryptographic handshake messages from the server and client.

The Destination Connection ID field in a Handshake packet contains a connection ID that is chosen by the recipient of the packet; the Source Connection ID includes the connection ID that the sender of the packet wishes to use (see Section 4.7).

The first Handshake packet sent by a server contains a randomized packet number. This value is increased for each subsequent packet sent by the server as described in Section 4.8. The client increments the packet number from its previous packet by one for each Handshake packet that it sends (which might be an Initial, 0-RTT Protected, or Handshake packet).

Servers MUST NOT send more than three Handshake packets without receiving a packet from a verified source address. Source addresses can be verified through an address validation token, receipt of the final cryptographic message from the client, or by receiving a valid PATH_RESPONSE frame from the client.

If the server expects to generate more than three Handshake packets in response to an Initial packet, it SHOULD include a PATH_CHALLENGE frame in each Handshake packet that it sends. After receiving at least one valid PATH_RESPONSE frame, the server can send its remaining Handshake packets. Servers can instead perform address validation using a Retry packet; this requires less state on the server, but could involve additional computational effort depending on implementation choices.

The payload of this packet contains STREAM frames and could contain PADDING, ACK, PATH_CHALLENGE, or PATH_RESPONSE frames. Handshake

packets MAY contain CONNECTION_CLOSE frames if the handshake is unsuccessful.

4.5. Protected Packets

Packets that are protected with 0-RTT keys are sent with long headers; all packets protected with 1-RTT keys are sent with short headers. The different packet types explicitly indicate the encryption level and therefore the keys that are used to remove packet protection.

Packets protected with 0-RTT keys use a type value of 0x7C. The connection ID fields for a 0-RTT packet MUST match the values used in the Initial packet (Section 4.4.1).

The client can send 0-RTT packets after receiving a Handshake packet (Section 4.4.3), if that packet does not complete the handshake. Even if the client receives a different connection ID in the Handshake packet, it MUST continue to use the same Destination Connection ID for 0-RTT packets, see Section 4.7.

The version field for protected packets is the current QUIC version.

The packet number field contains a packet number, which increases with each packet sent, see Section 4.8 for details.

The payload is protected using authenticated encryption. [QUIC-TLS] describes packet protection in detail. After decryption, the plaintext consists of a sequence of frames, as described in Section 5.

4.6. Coaslescing Packets

A sender can coalesce multiple QUIC packets (typically a Cryptographic Handshake packet and a Protected packet) into one UDP datagram. This can reduce the number of UDP datagrams needed to send application data during the handshake and immediately afterwards. A packet with a short header does not include a length, so it has to be the last packet included in a UDP datagram.

The sender MUST NOT coalesce QUIC packets belonging to different QUIC connections into a single UDP datagram.

Every QUIC packet that is coalesced into a single UDP datagram is separate and complete. Though the values of some fields in the packet header might be redundant, no fields are omitted. The receiver of coalesced QUIC packets MUST individually process each

QUIC packet and separately acknowledge them, as if they were received as the payload of different UDP datagrams.

4.7. Connection ID

A connection ID is used to ensure consistent routing of packets. The long header contains two connection IDs: the Destination Connection ID is chosen by the recipient of the packet and is used to provide consistent routing; the Source Connection ID is used to set the Destination Connection ID used by the peer.

During the handshake, packets with the long header are used to establish the connection ID that each endpoint uses. Each endpoint uses the Source Connection ID field to specify the connection ID that is used in the Destination Connection ID field of packets being sent to them. Upon receiving a packet, each endpoint sets the Destination Connection ID it sends to match the value of the Source Connection ID that they receive.

During the handshake, an endpoint might receive multiple packets with the long header, and thus be given multiple opportunities to update the Destination Connection ID it sends. A client MUST only change the value it sends in the Destination Connection ID in response to the first packet of each type it receives from the server (Retry or Handshake); a server MUST set its value based on the Initial packet. Any additional changes are not permitted; if subsequent packets of those types include a different Source Connection ID, they MUST be discarded. This avoids problems that might arise from stateless processing of multiple Initial packets producing different connection IDs.

Short headers only include the Destination Connection ID and omit the explicit length. The length of the Destination Connection ID field is expected to be known to endpoints.

Endpoints using a connection-ID based load balancer could agree with the load balancer on a fixed or minimum length and on an encoding for connection IDs. This fixed portion could encode an explicit length, which allows the entire connection ID to vary in length and still be used by the load balancer.

The very first packet sent by a client includes a random value for Destination Connection ID. The same value MUST be used for all 0-RTT packets sent on that connection (Section 4.5). This randomized value is used to determine the handshake packet protection keys (see Section 5.3.2 of [QUIC-TLS]).

A Version Negotiation (Section 4.3) packet MUST use both connection IDs selected by the client, swapped to ensure correct routing toward the client.

The connection ID can change over the lifetime of a connection, especially in response to connection migration (Section 6.8). NEW_CONNECTION_ID frames (Section 7.13) are used to provide new connection ID values.

4.8. Packet Numbers

The packet number is an integer in the range 0 to 2^62-1. The value is used in determining the cryptographic nonce for packet encryption. Each endpoint maintains a separate packet number for sending and receiving. The packet number for sending MUST increase by at least one after sending any packet, unless otherwise specified (see Section 4.8.1).

A QUIC endpoint MUST NOT reuse a packet number within the same connection (that is, under the same cryptographic keys). If the packet number for sending reaches 2^62 - 1, the sender MUST close the connection without sending a CONNECTION_CLOSE frame or any further packets; a server MAY send a Stateless Reset (Section 6.9.4) in response to further packets that it receives.

For the packet header, the number of bits required to represent the packet number are reduced by including only the least significant bits of the packet number. The actual packet number for each packet is reconstructed at the receiver based on the largest packet number received on a successfully authenticated packet.

A packet number is decoded by finding the packet number value that is closest to the next expected packet. The next expected packet is the highest received packet number plus one. For example, if the highest successfully authenticated packet had a packet number of 0xaa82f30e, then a packet containing a 16-bit value of 0x1f94 will be decoded as 0xaa831f94.

The sender MUST use a packet number size able to represent more than twice as large a range than the difference between the largest acknowledged packet and packet number being sent. A peer receiving the packet will then correctly decode the packet number, unless the packet is delayed in transit such that it arrives after many higher-numbered packets have been received. An endpoint SHOULD use a large enough packet number encoding to allow the packet number to be recovered even if the packet arrives after packets that are sent afterwards.

As a result, the size of the packet number encoding is at least one more than the base 2 logarithm of the number of contiguous unacknowledged packet numbers, including the new packet.

For example, if an endpoint has received an acknowledgment for packet 0x6afa2f, sending a packet with a number of 0x6b4264 requires a 16-bit or larger packet number encoding; whereas a 32-bit packet number is needed to send a packet with a number of 0x6bc107.

A Version Negotiation packet (Section 4.3) does not include a packet number. The Retry packet (Section 4.4.2) has special rules for populating the packet number field.

4.8.1. Initial Packet Number

The initial value for packet number MUST be selected randomly from a range between 0 and 2^32 - 1025 (inclusive). This value is selected so that Initial and Handshake packets exercise as many possible values for the Packet Number field as possible.

Limiting the range allows both for loss of packets and for any stateless exchanges. Packet numbers are incremented for subsequent packets, but packet loss and stateless handling can both mean that the first packet sent by an endpoint isn't necessarily the first packet received by its peer. The first packet received by a peer cannot be 2^32 or greater or the recipient will incorrectly assume a packet number that is 2^32 values lower and discard the packet.

Use of a secure random number generator [RFC4086] is not necessary for generating the initial packet number, nor is it necessary that the value be uniformly distributed.

5. Frames and Frame Types

The payload of all packets, after removing packet protection, consists of a sequence of frames, as shown in Figure 4. Version Negotiation and Stateless Reset do not contain frames.

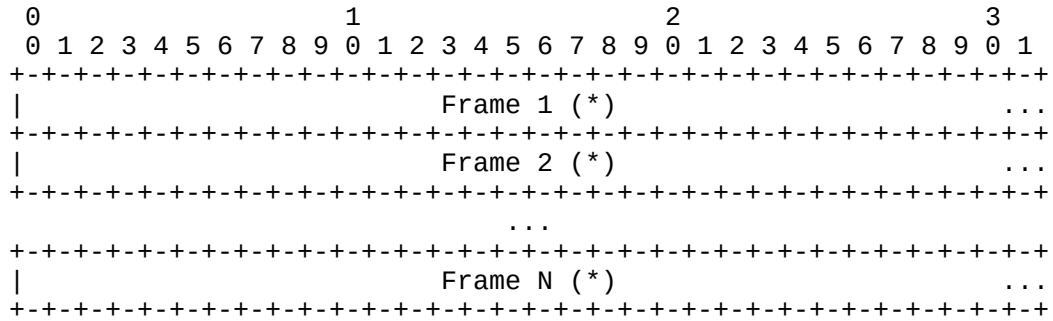


Figure 4: Contents of Protected Payload

Protected payloads MUST contain at least one frame, and MAY contain multiple frames and multiple frame types.

Frames MUST fit within a single QUIC packet and MUST NOT span a QUIC packet boundary. Each frame begins with a Frame Type byte, indicating its type, followed by additional type-dependent fields:

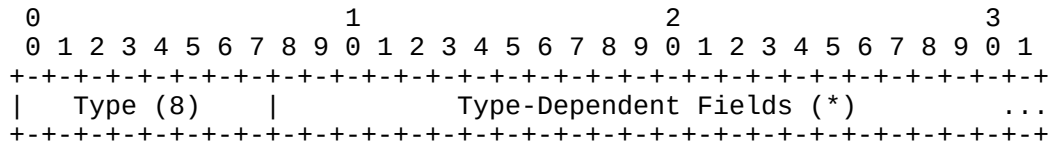


Figure 5: Generic Frame Layout

Frame types are listed in Table 3. Note that the Frame Type byte in STREAM frames is used to carry other frame-specific flags. For all other frames, the Frame Type byte simply identifies the frame. These frames are explained in more detail as they are referenced later in the document.

Type Value	Frame Type Name	Definition
0x00	PADDING	Section 7.2
0x01	RST_STREAM	Section 7.3
0x02	CONNECTION_CLOSE	Section 7.4
0x03	APPLICATION_CLOSE	Section 7.5
0x04	MAX_DATA	Section 7.6
0x05	MAX_STREAM_DATA	Section 7.7
0x06	MAX_STREAM_ID	Section 7.8
0x07	PING	Section 7.9
0x08	BLOCKED	Section 7.10
0x09	STREAM_BLOCKED	Section 7.11
0x0a	STREAM_ID_BLOCKED	Section 7.12
0x0b	NEW_CONNECTION_ID	Section 7.13
0x0c	STOP_SENDING	Section 7.14
0x0d	ACK	Section 7.15
0x0e	PATH_CHALLENGE	Section 7.16
0x0f	PATH_RESPONSE	Section 7.17
0x10 - 0x17	STREAM	Section 7.18

Table 3: Frame Types

6. Life of a Connection

A QUIC connection is a single conversation between two QUIC endpoints. QUIC's connection establishment intertwines version negotiation with the cryptographic and transport handshakes to reduce connection establishment latency, as described in Section 6.3. Once established, a connection may migrate to a different IP or port at either endpoint, due to NAT rebinding or mobility, as described in

Section 6.8. Finally a connection may be terminated by either endpoint, as described in Section 6.9.

6.1. Matching Packets to Connections

Incoming packets are classified on receipt. Packets can either be associated with an existing connection, or - for servers - potentially create a new connection.

Hosts try to associate a packet with an existing connection. If the packet has a Destination Connection ID corresponding to an existing connection, QUIC processes that packet accordingly. Note that a NEW_CONNECTION_ID frame (Section 7.13) would associate more than one connection ID with a connection.

If the Destination Connection ID is zero length and the packet matches the address/port tuple of a connection where the host did not require connection IDs, QUIC processes the packet as part of that connection. Endpoints MUST drop packets with zero-length Destination Connection ID fields if they do not correspond to a single connection.

6.1.1. Client Packet Handling

Valid packets sent to clients always include a Destination Connection ID that matches the value the client selects. Clients that choose to receive zero-length connection IDs can use the address/port tuple to identify a connection. Packets that don’t match an existing connection MAY be discarded.

Due to packet reordering or loss, clients might receive packets for a connection that are encrypted with a key it has not yet computed. Clients MAY drop these packets, or MAY buffer them in anticipation of later packets that allow it to compute the key.

If a client receives a packet that has an unsupported version, it MUST discard that packet.

6.1.2. Server Packet Handling

If a server receives a packet that has an unsupported version and sufficient length to be an Initial packet for some version supported by the server, it SHOULD send a Version Negotiation packet as described in Section 6.2.1. Servers MAY rate control these packets to avoid storms of Version Negotiation packets.

The first packet for an unsupported version can use different semantics and encodings for any version-specific field. In

particular, different packet protection keys might be used for different versions. Servers that do not support a particular version are unlikely to be able to decrypt the content of the packet. Servers SHOULD NOT attempt to decode or decrypt a packet from an unknown version, but instead send a Version Negotiation packet, provided that the packet is sufficiently long.

Servers MUST drop other packets that contain unsupported versions.

Packets with a supported version, or no version field, are matched to a connection as described in Section 6.1. If not matched, the server continues below.

If the packet is an Initial packet fully conforming with the specification, the server proceeds with the handshake (Section 6.3). This commits the server to the version that the client selected.

If a server isn't currently accepting any new connections, it SHOULD send a Handshake packet containing a CONNECTION_CLOSE frame with error code SERVER_BUSY.

If the packet is a 0-RTT packet, the server MAY buffer a limited number of these packets in anticipation of a late-arriving Initial Packet. Clients are forbidden from sending Handshake packets prior to receiving a server response, so servers SHOULD ignore any such packets.

Servers MUST drop incoming packets under all other circumstances. They SHOULD send a Stateless Reset (Section 6.9.4) if a connection ID is present in the header.

6.2. Version Negotiation

Version negotiation ensures that client and server agree to a QUIC version that is mutually supported. A server sends a Version Negotiation packet in response to each packet that might initiate a new connection, see Section 6.1 for details.

The size of the first packet sent by a client will determine whether a server sends a Version Negotiation packet. Clients that support multiple QUIC versions SHOULD pad their Initial packets to reflect the largest minimum Initial packet size of all their versions. This ensures that that the server responds if there are any mutually supported versions.

6.2.1. Sending Version Negotiation Packets

If the version selected by the client is not acceptable to the server, the server responds with a Version Negotiation packet (see Section 4.3). This includes a list of versions that the server will accept.

This system allows a server to process packets with unsupported versions without retaining state. Though either the Initial packet or the Version Negotiation packet that is sent in response could be lost, the client will send new packets until it successfully receives a response or it abandons the connection attempt.

6.2.2. Handling Version Negotiation Packets

When the client receives a Version Negotiation packet, it first checks that the Destination and Source Connection ID fields match the Source and Destination Connection ID fields in a packet that the client sent. If this check fails, the packet MUST be discarded.

Once the Version Negotiation packet is determined to be valid, the client then selects an acceptable protocol version from the list provided by the server. The client then attempts to create a connection using that version. Though the contents of the Initial packet the client sends might not change in response to version negotiation, a client MUST increase the packet number it uses on every packet it sends. Packets MUST continue to use long headers and MUST include the new negotiated protocol version.

The client MUST use the long header format and include its selected version on all packets until it has 1-RTT keys and it has received a packet from the server which is not a Version Negotiation packet.

A client MUST NOT change the version it uses unless it is in response to a Version Negotiation packet from the server. Once a client receives a packet from the server which is not a Version Negotiation packet, it MUST discard other Version Negotiation packets on the same connection. Similarly, a client MUST ignore a Version Negotiation packet if it has already received and acted on a Version Negotiation packet.

A client MUST ignore a Version Negotiation packet that lists the client’s chosen version.

Version negotiation packets have no cryptographic protection. The result of the negotiation MUST be revalidated as part of the cryptographic handshake (see Section 6.4.4).

6.2.3. Using Reserved Versions

For a server to use a new version in the future, clients must correctly handle unsupported versions. To help ensure this, a server SHOULD include a reserved version (see Section 3) while generating a Version Negotiation packet.

The design of version negotiation permits a server to avoid maintaining state for packets that it rejects in this fashion. The validation of version negotiation (see Section 6.4.4) only validates the result of version negotiation, which is the same no matter which reserved version was sent. A server MAY therefore send different reserved version numbers in the Version Negotiation Packet and in its transport parameters.

A client MAY send a packet using a reserved version number. This can be used to solicit a list of supported versions from a server.

6.3. Cryptographic and Transport Handshake

QUIC relies on a combined cryptographic and transport handshake to minimize connection establishment latency. QUIC allocates stream 0 for the cryptographic handshake. Version 0x00000001 of QUIC uses TLS 1.3 as described in [QUIC-TLS]; a different QUIC version number could indicate that a different cryptographic handshake protocol is in use.

QUIC provides this stream with reliable, ordered delivery of data. In return, the cryptographic handshake provides QUIC with:

- authenticated key exchange, where
  - a server is always authenticated,
  - a client is optionally authenticated,
  - every connection produces distinct and unrelated keys,
  - keying material is usable for packet protection for both 0-RTT and 1-RTT packets, and
  - 1-RTT keys have forward secrecy
- authenticated values for the transport parameters of the peer (see Section 6.4)
- authenticated confirmation of version negotiation (see Section 6.4.4)

- authenticated negotiation of an application protocol (TLS uses ALPN [RFC7301] for this purpose)

- for the server, the ability to carry data that provides assurance that the client can receive packets that are addressed with the transport address that is claimed by the client (see Section 6.6)

The initial cryptographic handshake message MUST be sent in a single packet. Any second attempt that is triggered by address validation MUST also be sent within a single packet. This avoids having to reassemble a message from multiple packets. Reassembling messages requires that a server maintain state prior to establishing a connection, exposing the server to a denial of service risk.

The first client packet of the cryptographic handshake protocol MUST fit within a 1232 octet QUIC packet payload. This includes overheads that reduce the space available to the cryptographic handshake protocol.

Details of how TLS is integrated with QUIC is provided in more detail in [QUIC-TLS].

6.4. Transport Parameters

During connection establishment, both endpoints make authenticated declarations of their transport parameters. These declarations are made unilaterally by each endpoint. Endpoints are required to comply with the restrictions implied by these parameters; the description of each parameter includes rules for its handling.

The format of the transport parameters is the TransportParameters struct from Figure 6. This is described using the presentation language from Section 3 of [I-D.ietf-tls-tls13].

```
uint32 QuicVersion;

enum {
   initial_max_stream_data(0),
   initial_max_data(1),
   initial_max_stream_id_bidi(2),
   idle_timeout(3),
   max_packet_size(5),
   stateless_reset_token(6),
   ack_delay_exponent(7),
   initial_max_stream_id_uni(8),
   (65535)
} TransportParameterId;

struct {
   TransportParameterId parameter;
   opaque value<0..2^16-1>;
} TransportParameter;

struct {
   select (Handshake.msg_type) {
      case client_hello:
         QuicVersion initial_version;

      case encrypted_extensions:
         QuicVersion negotiated_version;
         QuicVersion supported_versions<4..2^8-4>;
   };
   TransportParameter parameters<22..2^16-1>;
} TransportParameters;
```

Figure 6: Definition of TransportParameters

The "extension_data" field of the quic_transport_parameters extension defined in [QUIC-TLS] contains a TransportParameters value. TLS encoding rules are therefore used to encode the transport parameters.

QUIC encodes transport parameters into a sequence of octets, which are then included in the cryptographic handshake. Once the handshake completes, the transport parameters declared by the peer are available. Each endpoint validates the value provided by its peer. In particular, version negotiation MUST be validated (see Section 6.4.4) before the connection establishment is considered properly complete.

Definitions for each of the defined transport parameters are included in Section 6.4.1. Any given parameter MUST appear at most once in a given transport parameters extension. An endpoint MUST treat receipt

of duplicate transport parameters as a connection error of type TRANSPORT_PARAMETER_ERROR.

6.4.1. Transport Parameter Definitions

An endpoint MUST include the following parameters in its encoded TransportParameters:

initial_max_stream_data (0x0000): The initial stream maximum data parameter contains the initial value for the maximum data that can be sent on any newly created stream. This parameter is encoded as an unsigned 32-bit integer in units of octets. This is equivalent to an implicit MAX_STREAM_DATA frame (Section 7.7) being sent on all streams immediately after opening.

initial_max_data (0x0001): The initial maximum data parameter contains the initial value for the maximum amount of data that can be sent on the connection. This parameter is encoded as an unsigned 32-bit integer in units of octets. This is equivalent to sending a MAX_DATA (Section 7.6) for the connection immediately after completing the handshake.

idle_timeout (0x0003): The idle timeout is a value in seconds that is encoded as an unsigned 16-bit integer. The maximum value is 600 seconds (10 minutes).

An endpoint MAY use the following transport parameters:

initial_max_streams_bidi (0x0002): The initial maximum bidirectional streams parameter contains the initial maximum number of application-owned bidirectional streams the peer may initiate, encoded as an unsigned 16-bit integer. If this parameter is absent or zero, application-owned bidirectional streams cannot be created until a MAX_STREAM_ID frame is sent. Note that a value of 0 does not prevent the cryptographic handshake stream (that is, stream 0) from being used. Setting this parameter is equivalent to sending a MAX_STREAM_ID (Section 7.8) immediately after completing the handshake containing the corresponding Stream ID. For example, a value of 0x05 would be equivalent to receiving a MAX_STREAM_ID containing 20 when received by a client or 17 when received by a server.

initial_max_stream_id_uni (0x0008): The initial maximum unidirectional streams parameter contains the initial maximum number of application-owned unidirectional streams the peer may initiate, encoded as an unsigned 16-bit integer. If this parameter is absent or zero, unidirectional streams cannot be created until a MAX_STREAM_ID frame is sent. Setting this

parameter is equivalent to sending a MAX_STREAM_ID (Section 7.8) immediately after completing the handshake containing the corresponding Stream ID. For example, a value of 0x05 would be equivalent to receiving a MAX_STREAM_ID containing 18 when received by a client or 19 when received by a server.

max_packet_size (0x0005): The maximum packet size parameter places a limit on the size of packets that the endpoint is willing to receive, encoded as an unsigned 16-bit integer. This indicates that packets larger than this limit will be dropped. The default for this parameter is the maximum permitted UDP payload of 65527. Values below 1200 are invalid. This limit only applies to protected packets (Section 4.5).

ack_delay_exponent (0x0007): An 8-bit unsigned integer value indicating an exponent used to decode the ACK Delay field in the ACK frame, see Section 7.15. If this value is absent, a default value of 3 is assumed (indicating a multiplier of 8). The default value is also used for ACK frames that are sent in Initial, Handshake, and Retry packets. Values above 20 are invalid.

A server MAY include the following transport parameters:

stateless_reset_token (0x0006): The Stateless Reset Token is used in verifying a stateless reset, see Section 6.9.4. This parameter is a sequence of 16 octets.

A client MUST NOT include a stateless reset token. A server MUST treat receipt of a stateless_reset_token transport parameter as a connection error of type TRANSPORT_PARAMETER_ERROR.

6.4.2. Values of Transport Parameters for 0-RTT

A client that attempts to send 0-RTT data MUST remember the transport parameters used by the server. The transport parameters that the server advertises during connection establishment apply to all connections that are resumed using the keying material established during that handshake. Remembered transport parameters apply to the new connection until the handshake completes and new transport parameters from the server can be provided.

A server can remember the transport parameters that it advertised, or store an integrity-protected copy of the values in the ticket and recover the information when accepting 0-RTT data. A server uses the transport parameters in determining whether to accept 0-RTT data.

A server MAY accept 0-RTT and subsequently provide different values for transport parameters for use in the new connection. If 0-RTT

data is accepted by the server, the server MUST NOT reduce any limits or alter any values that might be violated by the client with its 0-RTT data. In particular, a server that accepts 0-RTT data MUST NOT set values for initial_max_data or initial_max_stream_data that are smaller than the remembered value of those parameters. Similarly, a server MUST NOT reduce the value of initial_max_stream_id_bidi or initial_max_stream_id_uni.

Omitting or setting a zero value for certain transport parameters can result in 0-RTT data being enabled, but not usable. The following transport parameters SHOULD be set to non-zero values for 0-RTT: initial_max_stream_id_bidi, initial_max_stream_id_uni, initial_max_data, initial_max_stream_data.

A server MUST reject 0-RTT data or even abort a handshake if the implied values for transport parameters cannot be supported.

6.4.3. New Transport Parameters

New transport parameters can be used to negotiate new protocol behavior. An endpoint MUST ignore transport parameters that it does not support. Absence of a transport parameter therefore disables any optional protocol feature that is negotiated using the parameter.

New transport parameters can be registered according to the rules in Section 13.1.

6.4.4. Version Negotiation Validation

Though the cryptographic handshake has integrity protection, two forms of QUIC version downgrade are possible. In the first, an attacker replaces the QUIC version in the Initial packet. In the second, a fake Version Negotiation packet is sent by an attacker. To protect against these attacks, the transport parameters include three fields that encode version information. These parameters are used to retroactively authenticate the choice of version (see Section 6.2).

The cryptographic handshake provides integrity protection for the negotiated version as part of the transport parameters (see Section 6.4). As a result, attacks on version negotiation by an attacker can be detected.

The client includes the initial_version field in its transport parameters. The initial_version is the version that the client initially attempted to use. If the server did not send a Version Negotiation packet Section 4.3, this will be identical to the negotiated_version field in the server transport parameters.

A server that processes all packets in a stateful fashion can remember how version negotiation was performed and validate the initial_version value.

A server that does not maintain state for every packet it receives (i.e., a stateless server) uses a different process. If the initial_version matches the version of QUIC that is in use, a stateless server can accept the value.

If the initial_version is different from the version of QUIC that is in use, a stateless server MUST check that it would have sent a Version Negotiation packet if it had received a packet with the indicated initial_version. If a server would have accepted the version included in the initial_version and the value differs from the QUIC version that is in use, the server MUST terminate the connection with a VERSION_NEGOTIATION_ERROR error.

The server includes both the version of QUIC that is in use and a list of the QUIC versions that the server supports.

The negotiated_version field is the version that is in use. This MUST be set by the server to the value that is on the Initial packet that it accepts (not an Initial packet that triggers a Retry or Version Negotiation packet). A client that receives a negotiated_version that does not match the version of QUIC that is in use MUST terminate the connection with a VERSION_NEGOTIATION_ERROR error code.

The server includes a list of versions that it would send in any version negotiation packet (Section 4.3) in the supported_versions field. The server populates this field even if it did not send a version negotiation packet.

The client validates that the negotiated_version is included in the supported_versions list and - if version negotiation was performed - that it would have selected the negotiated version. A client MUST terminate the connection with a VERSION_NEGOTIATION_ERROR error code if the current QUIC version is not listed in the supported_versions list. A client MUST terminate with a VERSION_NEGOTIATION_ERROR error code if version negotiation occurred but it would have selected a different version based on the value of the supported_versions list.

When an endpoint accepts multiple QUIC versions, it can potentially interpret transport parameters as they are defined by any of the QUIC versions it supports. The version field in the QUIC packet header is authenticated using transport parameters. The position and the format of the version fields in transport parameters MUST either be identical across different QUIC versions, or be unambiguously

different to ensure no confusion about their interpretation. One way that a new format could be introduced is to define a TLS extension with a different codepoint.

6.5. Stateless Retries

A server can process an initial cryptographic handshake messages from a client without committing any state. This allows a server to perform address validation (Section 6.6, or to defer connection establishment costs.

A server that generates a response to an initial packet without retaining connection state MUST use the Retry packet (Section 4.4.2). This packet causes a client to reset its transport state and to continue the connection attempt with new connection state while maintaining the state of the cryptographic handshake.

A server MUST NOT send multiple Retry packets in response to a client handshake packet. Thus, any cryptographic handshake message that is sent MUST fit within a single packet.

In TLS, the Retry packet type is used to carry the HelloRetryRequest message.

6.6. Proof of Source Address Ownership

Transport protocols commonly spend a round trip checking that a client owns the transport address (IP and port) that it claims. Verifying that a client can receive packets sent to its claimed transport address protects against spoofing of this information by malicious clients.

This technique is used primarily to avoid QUIC from being used for traffic amplification attack. In such an attack, a packet is sent to a server with spoofed source address information that identifies a victim. If a server generates more or larger packets in response to that packet, the attacker can use the server to send more data toward the victim than it would be able to send on its own.

Several methods are used in QUIC to mitigate this attack. Firstly, the initial handshake packet is padded to at least 1200 octets. This allows a server to send a similar amount of data without risking causing an amplification attack toward an unproven remote address.

A server eventually confirms that a client has received its messages when the cryptographic handshake successfully completes. This might be insufficient, either because the server wishes to avoid the computational cost of completing the handshake, or it might be that

the size of the packets that are sent during the handshake is too large. This is especially important for 0-RTT, where the server might wish to provide application data traffic - such as a response to a request - in response to the data carried in the early data from the client.

To send additional data prior to completing the cryptographic handshake, the server then needs to validate that the client owns the address that it claims.

Source address validation is therefore performed during the establishment of a connection. TLS provides the tools that support the feature, but basic validation is performed by the core transport protocol.

A different type of source address validation is performed after a connection migration, see Section 6.7.

6.6.1. Client Address Validation Procedure

QUIC uses token-based address validation. Any time the server wishes to validate a client address, it provides the client with a token. As long as the token cannot be easily guessed (see Section 6.6.3), if the client is able to return that token, it proves to the server that it received the token.

During the processing of the cryptographic handshake messages from a client, TLS will request that QUIC make a decision about whether to proceed based on the information it has. TLS will provide QUIC with any token that was provided by the client. For an initial packet, QUIC can decide to abort the connection, allow it to proceed, or request address validation.

If QUIC decides to request address validation, it provides the cryptographic handshake with a token. The contents of this token are consumed by the server that generates the token, so there is no need for a single well-defined format. A token could include information about the claimed client address (IP and port), a timestamp, and any other supplementary information the server will need to validate the token in the future.

The cryptographic handshake is responsible for enacting validation by sending the address validation token to the client. A legitimate client will include a copy of the token when it attempts to continue the handshake. The cryptographic handshake extracts the token then asks QUIC a second time whether the token is acceptable. In response, QUIC can either abort the connection or permit it to proceed.

A connection MAY be accepted without address validation - or with only limited validation - but a server SHOULD limit the data it sends toward an unvalidated address. Successful completion of the cryptographic handshake implicitly provides proof that the client has received packets from the server.

6.6.2. Address Validation on Session Resumption

A server MAY provide clients with an address validation token during one connection that can be used on a subsequent connection. Address validation is especially important with 0-RTT because a server potentially sends a significant amount of data to a client in response to 0-RTT data.

A different type of token is needed when resuming. Unlike the token that is created during a handshake, there might be some time between when the token is created and when the token is subsequently used. Thus, a resumption token SHOULD include an expiration time. It is also unlikely that the client port number is the same on two different connections; validating the port is therefore unlikely to be successful.

This token can be provided to the cryptographic handshake immediately after establishing a connection. QUIC might also generate an updated token if significant time passes or the client address changes for any reason (see Section 6.8). The cryptographic handshake is responsible for providing the client with the token. In TLS the token is included in the ticket that is used for resumption and 0-RTT, which is carried in a NewSessionTicket message.

6.6.3. Address Validation Token Integrity

An address validation token MUST be difficult to guess. Including a large enough random value in the token would be sufficient, but this depends on the server remembering the value it sends to clients.

A token-based scheme allows the server to offload any state associated with validation to the client. For this design to work, the token MUST be covered by integrity protection against modification or falsification by clients. Without integrity protection, malicious clients could generate or guess values for tokens that would be accepted by the server. Only the server requires access to the integrity protection key for tokens.

In TLS the address validation token is often bundled with the information that TLS requires, such as the resumption secret. In this case, adding integrity protection can be delegated to the cryptographic handshake protocol, avoiding redundant protection. If

integrity protection is delegated to the cryptographic handshake, an integrity failure will result in immediate cryptographic handshake failure. If integrity protection is performed by QUIC, QUIC MUST abort the connection if the integrity check fails with a PROTOCOL_VIOLATION error code.

6.7. Path Validation

Path validation is used by an endpoint to verify reachability of a peer over a specific path. That is, it tests reachability between a specific local address and a specific peer address, where an address is the two-tuple of IP address and port. Path validation tests that packets can be both sent to and received from a peer.

Path validation is used during connection migration (see Section 6.8) by the migrating endpoint to verify reachability of a peer from a new local address. Path validation is also used by the peer to verify that the migrating endpoint is able to receive packets sent to its new address. That is, that the packets received from the migrating endpoint do not carry a spoofed source address.

Path validation can be used at any time by either endpoint. For instance, an endpoint might check that a peer is still in possession of its address after a period of quiescence.

Path validation is not designed as a NAT traversal mechanism. Though the mechanism described here might be effective for the creation of NAT bindings that support NAT traversal, the expectation is that one or other peer is able to receive packets without first having sent a packet on that path. Effective NAT traversal needs additional synchronization mechanisms that are not provided here.

An endpoint MAY bundle PATH_CHALLENGE and PATH_RESPONSE frames that are used for path validation with other frames. For instance, an endpoint may pad a packet carrying a PATH_CHALLENGE for PMTU discovery, or an endpoint may bundle a PATH_RESPONSE with its own PATH_CHALLENGE.

6.7.1. Initiation

To initiate path validation, an endpoint sends a PATH_CHALLENGE frame containing a random payload on the path to be validated.

An endpoint MAY send additional PATH_CHALLENGE frames to handle packet loss. An endpoint SHOULD NOT send a PATH_CHALLENGE more frequently than it would an Initial packet, ensuring that connection migration is no more load on a new path than establishing a new connection.

The endpoint MUST use fresh random data in every PATH_CHALLENGE frame so that it can associate the peer's response with the causative PATH_CHALLENGE.

6.7.2. Response

On receiving a PATH_CHALLENGE frame, an endpoint MUST respond immediately by echoing the data contained in the PATH_CHALLENGE frame in a PATH_RESPONSE frame, with the following stipulation. Since a PATH_CHALLENGE might be sent from a spoofed address, an endpoint MAY limit the rate at which it sends PATH_RESPONSE frames and MAY silently discard PATH_CHALLENGE frames that would cause it to respond at a higher rate.

To ensure that packets can be both sent to and received from the peer, the PATH_RESPONSE MUST be sent on the same path as the triggering PATH_CHALLENGE: from the same local address on which the PATH_CHALLENGE was received, to the same remote address from which the PATH_CHALLENGE was received.

6.7.3. Completion

A new address is considered valid when a PATH_RESPONSE frame is received containing data that was sent in a previous PATH_CHALLENGE. Receipt of an acknowledgment for a packet containing a PATH_CHALLENGE frame is not adequate validation, since the acknowledgment can be spoofed by a malicious peer.

For path validation to be successful, a PATH_RESPONSE frame MUST be received from the same remote address to which the corresponding PATH_CHALLENGE was sent. If a PATH_RESPONSE frame is received from a different remote address than the one to which the PATH_CHALLENGE was sent, path validation is considered to have failed, even if the data matches that sent in the PATH_CHALLENGE.

Additionally, the PATH_RESPONSE frame MUST be received on the same local address from which the corresponding PATH_CHALLENGE was sent. If a PATH_RESPONSE frame is received on a different local address than the one from which the PATH_CHALLENGE was sent, path validation is considered to have failed, even if the data matches that sent in the PATH_CHALLENGE. Thus, the endpoint considers the path to be valid when a PATH_RESPONSE frame is received on the same path with the same payload as the PATH_CHALLENGE frame.

6.7.4. Abandonment

An endpoint SHOULD abandon path validation after sending some number of PATH_CHALLENGE frames or after some time has passed. When setting this timer, implementations are cautioned that the new path could have a longer round-trip time than the original.

Note that the endpoint might receive packets containing other frames on the new path, but a PATH_RESPONSE frame with appropriate data is required for path validation to succeed.

If path validation fails, the path is deemed unusable. This does not necessarily imply a failure of the connection - endpoints can continue sending packets over other paths as appropriate. If no paths are available, an endpoint can wait for a new path to become available or close the connection.

A path validation might be abandoned for other reasons besides failure. Primarily, this happens if a connection migration to a new path is initiated while a path validation on the old path is in progress.

6.8. Connection Migration

QUIC allows connections to survive changes to endpoint addresses (that is, IP address and/or port), such as those caused by a endpoint migrating to a new network. This section describes the process by which an endpoint migrates to a new address.

An endpoint MUST NOT initiate connection migration before the handshake is finished and the endpoint has 1-RTT keys.

This document limits migration of connections to new client addresses. Clients are responsible for initiating all migrations. Servers do not send non-probing packets (see Section 6.8.1) toward a client address until it sees a non-probing packet from that address. If a client receives packets from an unknown server address, the client MAY discard these packets. Migrating a connection to a new server address is left for future work.

6.8.1. Probing a New Path

An endpoint MAY probe for peer reachability from a new local address using path validation Section 6.7 prior to migrating the connection to the new local address. Failure of path validation simply means that the new path is not usable for this connection. Failure to validate a path does not cause the connection to end unless there are no valid alternative paths available.

An endpoint uses a new connection ID for probes sent from a new local address, see Section 6.8.5 for further discussion.

Receiving a PATH_CHALLENGE frame from a peer indicates that the peer is probing for reachability on a path. An endpoint sends a PATH_RESPONSE in response as per Section 6.7.

PATH_CHALLENGE, PATH_RESPONSE, and PADDING frames are "probing frames", and all other frames are "non-probing frames". A packet containing only probing frames is a "probing packet", and a packet containing any other frame is a "non-probing packet".

6.8.2. Initiating Connection Migration

A endpoint can migrate a connection to a new local address by sending packets containing frames other than probing frames from that address.

Each endpoint validates its peer's address during connection establishment. Therefore, a migrating endpoint can send to its peer knowing that the peer is willing to receive at the peer's current address. Thus an endpoint can migrate to a new local address without first validating the peer's address.

When migrating, the new path might not support the endpoint's current sending rate. Therefore, the endpoint resets its congestion controller, as described in Section 6.8.4.

Receiving acknowledgments for data sent on the new path serves as proof of the peer's reachability from the new address. Note that since acknowledgments may be received on any path, return reachability on the new path is not established. To establish return reachability on the new path, an endpoint MAY concurrently initiate path validation Section 6.7 on the new path.

6.8.3. Responding to Connection Migration

Receiving a packet from a new peer address containing a non-probing frame indicates that the peer has migrated to that address.

In response to such a packet, an endpoint MUST start sending subsequent packets to the new peer address and MUST initiate path validation (Section 6.7) to verify the peer's ownership of the unvalidated address.

An endpoint MAY send data to an unvalidated peer address, but it MUST protect against potential attacks as described in Section 6.8.3.1 and

Section 6.8.3.2. An endpoint MAY skip validation of a peer address if that address has been seen recently.

An endpoint only changes the address that it sends packets to in response to the highest-numbered non-probing packet. This ensures that an endpoint does not send packets to an old peer address in the case that it receives reordered packets.

After changing the address to which it sends non-probing packets, an endpoint could abandon any path validation for other addresses.

Receiving a packet from a new peer address might be the result of a NAT rebinding at the peer.

After verifying a new client address, the server SHOULD send new address validation tokens (Section 6.6) to the client.

6.8.3.1. Handling Address Spoofing by a Peer

It is possible that a peer is spoofing its source address to cause an endpoint to send excessive amounts of data to an unwilling host. If the endpoint sends significantly more data than the spoofing peer, connection migration might be used to amplify the volume of data that an attacker can generate toward a victim.

As described in Section 6.8.3, an endpoint is required to validate a peer's new address to confirm the peer's possession of the new address. Until a peer's address is deemed valid, an endpoint MUST limit the rate at which it sends data to this address. The endpoint MUST NOT send more than a minimum congestion window's worth of data per estimated round-trip time (kMinimumWindow, as defined in [QUIC-RECOVERY]). In the absence of this limit, an endpoint risks being used for a denial of service attack against an unsuspecting victim. Note that since the endpoint will not have any round-trip time measurements to this address, the estimate SHOULD be the default initial value (see [QUIC-RECOVERY]).

If an endpoint skips validation of a peer address as described in Section 6.8.3, it does not need to limit its sending rate.

6.8.3.2. Handling Address Spoofing by an On-path Attacker

An on-path attacker could cause a spurious connection migration by copying and forwarding a packet with a spoofed address such that it arrives before the original packet. The packet with the spoofed address will be seen to come from a migrating connection, and the original packet will be seen as a duplicate and dropped. After a spurious migration, validation of the source address will fail

because the entity at the source address does not have the necessary cryptographic keys to read or respond to the PATH_CHALLENGE frame that is sent to it even if it wanted to.

To protect the connection from failing due to such a spurious migration, an endpoint MUST revert to using the last validated peer address when validation of a new peer address fails.

If an endpoint has no state about the last validated peer address, it MUST close the connection silently by discarding all connection state. This results in new packets on the connection being handled generically. For instance, an endpoint MAY send a stateless reset in response to any further incoming packets.

Note that receipt of packets with higher packet numbers from the legitimate peer address will trigger another connection migration. This will cause the validation of the address of the spurious migration to be abandoned.

6.8.4. Loss Detection and Congestion Control

The capacity available on the new path might not be the same as the old path. Packets sent on the old path SHOULD NOT contribute to congestion control or RTT estimation for the new path.

On confirming a peer’s ownership of its new address, an endpoint SHOULD immediately reset the congestion controller and round-trip time estimator for the new path.

An endpoint MUST NOT return to the send rate used for the previous path unless it is reasonably sure that the previous send rate is valid for the new path. For instance, a change in the client’s port number is likely indicative of a rebinding in a middlebox and not a complete change in path. This determination likely depends on heuristics, which could be imperfect; if the new path capacity is significantly reduced, ultimately this relies on the congestion controller responding to congestion signals and reducing send rates appropriately.

There may be apparent reordering at the receiver when an endpoint sends data and probes from/to multiple addresses during the migration period, since the two resulting paths may have different round-trip times. A receiver of packets on multiple paths will still send ACK frames covering all received packets.

While multiple paths might be used during connection migration, a single congestion control context and a single loss recovery context (as described in [QUIC-RECOVERY]) may be adequate. A sender can make

exceptions for probe packets so that their loss detection is independent and does not unduly cause the congestion controller to reduce its sending rate. An endpoint might arm a separate alarm when a PATH_CHALLENGE is sent, which is disarmed when the corresponding PATH_RESPONSE is received. If the alarm fires before the PATH_RESPONSE is received, the endpoint might send a new PATH_CHALLENGE, and restart the alarm for a longer period of time.

6.8.5. Privacy Implications of Connection Migration

Using a stable connection ID on multiple network paths allows a passive observer to correlate activity between those paths. An endpoint that moves between networks might not wish to have their activity correlated by any entity other than a server. The NEW_CONNECTION_ID message can be sent by both endpoints to provide an unlinkable connection ID for use in case a peer wishes to explicitly break linkability between two points of network attachment.

An endpoint might need to send packets on multiple networks without receiving any response from its peer. To ensure that the endpoint is not linkable across each of these changes, a new connection ID and packet number gap are needed for each network. To support this, each endpoint sends multiple NEW_CONNECTION_ID messages. Each NEW_CONNECTION_ID is marked with a sequence number. Connection IDs MUST be used in the order in which they are numbered.

An endpoint that does not require the use of a connection ID should not request that its peer use a connection ID. Such an endpoint does not need to provide new connection IDs using the NEW_CONNECTION_ID frame.

An endpoint which wishes to break linkability upon changing networks MUST use the connection ID provided by its peer as well as incrementing the packet sequence number by an externally unpredictable value computed as described in Section 6.8.5.1. Packet number gaps are cumulative. An endpoint might skip connection IDs, but it MUST ensure that it applies the associated packet number gaps for connection IDs that it skips in addition to the packet number gap associated with the connection ID that it does use.

An endpoint that receives a packet that is marked with a new connection ID recovers the packet number by adding the cumulative packet number gap to its expected packet number. An endpoint MUST discard packets that contain a smaller gap than it advertised.

Clients MAY change connection ID at any time based on implementation-specific concerns. For example, after a period of network inactivity NAT rebinding might occur when the client begins sending data again.

A client might wish to reduce linkability by employing a new connection ID and source UDP port when sending traffic after a period of inactivity. Changing the UDP port from which it sends packets at the same time might cause the packet to appear as a connection migration. This ensures that the mechanisms that support migration are exercised even for clients that don't experience NAT rebindings or genuine migrations. Changing port number can cause a peer to reset its congestion state (see Section 6.8.4), so the port SHOULD only be changed infrequently.

An endpoint that receives a successfully authenticated packet with a previously unused connection ID MUST use the next available connection ID for any packets it sends to that address. To avoid changing connection IDs multiple times when packets arrive out of order, endpoints MUST change only in response to a packet that increases the largest received packet number. Failing to do this could allow for use of that connection ID to link activity on new paths. There is no need to move to a new connection ID if the address of a peer changes without also changing the connection ID.

For instance, a server might provide a packet number gap of 7 associated with a new connection ID. If the server received packet 10 using the previous connection ID, it should expect packets on the new connection ID to start at 18. A packet with the new connection ID and a packet number of 17 is discarded as being in error.

6.8.5.1. Packet Number Gap

In order to avoid linkage, the packet number gap MUST be externally indistinguishable from random. The packet number gap for a connection ID with sequence number is computed by encoding the sequence number as a 32-bit integer in big-endian format, and then computing:

```
Gap = HKDF-Expand-Label(packet_number_secret,
                        "QUIC packet sequence gap", sequence, 4)
```

The output of HKDF-Expand-Label is interpreted as a big-endian number. "packet_number_secret" is derived from the TLS key exchange, as described in Section 5.6 of [QUIC-TLS].

6.9. Connection Termination

Connections should remain open until they become idle for a pre-negotiated period of time. A QUIC connection, once established, can be terminated in one of three ways:

- idle timeout (Section 6.9.2)

- immediate close (Section 6.9.3)

- stateless reset (Section 6.9.4)

6.9.1. Closing and Draining Connection States

The closing and draining connection states exist to ensure that connections close cleanly and that delayed or reordered packets are properly discarded. These states SHOULD persist for three times the current Retransmission Timeout (RTO) interval as defined in [QUIC-RECOVERY].

An endpoint enters a closing period after initiating an immediate close (Section 6.9.3). While closing, an endpoint MUST NOT send packets unless they contain a CONNECTION_CLOSE or APPLICATION_CLOSE frame (see Section 6.9.3 for details).

In the closing state, only a packet containing a closing frame can be sent. An endpoint retains only enough information to generate a packet containing a closing frame and to identify packets as belonging to the connection. The connection ID and QUIC version is sufficient information to identify packets for a closing connection; an endpoint can discard all other connection state. An endpoint MAY retain packet protection keys for incoming packets to allow it to read and process a closing frame.

The draining state is entered once an endpoint receives a signal that its peer is closing or draining. While otherwise identical to the closing state, an endpoint in the draining state MUST NOT send any packets. Retaining packet protection keys is unnecessary once a connection is in the draining state.

An endpoint MAY transition from the closing period to the draining period if it can confirm that its peer is also closing or draining. Receiving a closing frame is sufficient confirmation, as is receiving a stateless reset. The draining period SHOULD end when the closing period would have ended. In other words, the endpoint can use the same end time, but cease retransmission of the closing packet.

Disposing of connection state prior to the end of the closing or draining period could cause delayed or reordered packets to be handled poorly. Endpoints that have some alternative means to ensure that late-arriving packets on the connection do not create QUIC state, such as those that are able to close the UDP socket, MAY use an abbreviated draining period which can allow for faster resource recovery. Servers that retain an open socket for accepting new connections SHOULD NOT exit the closing or draining period early.

Once the closing or draining period has ended, an endpoint SHOULD discard all connection state. This results in new packets on the connection being handled generically. For instance, an endpoint MAY send a stateless reset in response to any further incoming packets.

The draining and closing periods do not apply when a stateless reset (Section 6.9.4) is sent.

An endpoint is not expected to handle key updates when it is closing or draining. A key update might prevent the endpoint from moving from the closing state to draining, but it otherwise has no impact.

An endpoint could receive packets from a new source address, indicating a client connection migration (Section 6.8), while in the closing period. An endpoint in the closing state MUST strictly limit the number of packets it sends to this new address until the address is validated (see Section 6.7). A server in the closing state MAY instead choose to discard packets received from a new source address.

6.9.2. Idle Timeout

A connection that remains idle for longer than the idle timeout (see Section 6.4.1) is closed. A connection enters the draining state when the idle timeout expires.

The time at which an idle timeout takes effect won’t be perfectly synchronized on both endpoints. An endpoint that sends packets near the end of an idle period could have those packets discarded if its peer enters the draining state before the packet is received.

6.9.3. Immediate Close

An endpoint sends a closing frame, either CONNECTION_CLOSE or APPLICATION_CLOSE, to terminate the connection immediately. Either closing frame causes all streams to immediately become closed; open streams can be assumed to be implicitly reset.

After sending a closing frame, endpoints immediately enter the closing state. During the closing period, an endpoint that sends a closing frame SHOULD respond to any packet that it receives with another packet containing a closing frame. To minimize the state that an endpoint maintains for a closing connection, endpoints MAY send the exact same packet. However, endpoints SHOULD limit the number of packets they generate containing a closing frame. For instance, an endpoint could progressively increase the number of packets that it receives before sending additional packets or increase the time between packets.

Note: Allowing retransmission of a packet contradicts other advice in this document that recommends the creation of new packet numbers for every packet. Sending new packet numbers is primarily of advantage to loss recovery and congestion control, which are not expected to be relevant for a closed connection. Retransmitting the final packet requires less state.

After receiving a closing frame, endpoints enter the draining state. An endpoint that receives a closing frame MAY send a single packet containing a closing frame before entering the draining state, using a CONNECTION_CLOSE frame and a NO_ERROR code if appropriate. An endpoint MUST NOT send further packets, which could result in a constant exchange of closing frames until the closing period on either peer ended.

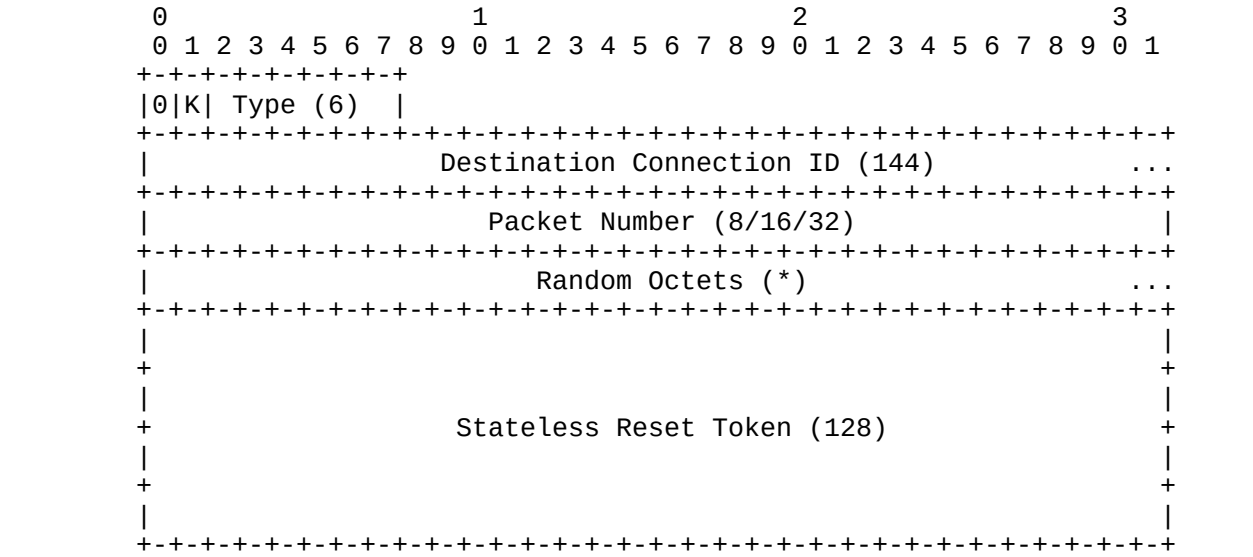
An immediate close can be used after an application protocol has arranged to close a connection. This might be after the application protocols negotiates a graceful shutdown. The application protocol exchanges whatever messages that are needed to cause both endpoints to agree to close the connection, after which the application requests that the connection be closed. The application protocol can use an APPLICATION_CLOSE message with an appropriate error code to signal closure.

6.9.4. Stateless Reset

A stateless reset is provided as an option of last resort for a server that does not have access to the state of a connection. A server crash or outage might result in clients continuing to send data to a server that is unable to properly continue the connection. A server that wishes to communicate a fatal connection error MUST use a closing frame if it has sufficient state to do so.

To support this process, the server sends a stateless_reset_token value during the handshake in the transport parameters. This value is protected by encryption, so only client and server know this value.

A server that receives packets that it cannot process sends a packet in the following layout:



This design ensures that a stateless reset packet is - to the extent possible - indistinguishable from a regular packet with a short header.

A server generates a random 18-octet Destination Connection ID field. For a client that depends on the server including a connection ID, this will mean that this value differs from previous packets. Ths results in two problems:

- The packet might not reach the client. If the Destination Connection ID is critical for routing toward the client, then this packet could be incorrectly routed. This causes the stateless reset to be ineffective in causing errors to be quickly detected and recovered. In this case, clients will need to rely on other methods - such as timers - to detect that the connection has failed.

- The randomly generated connection ID can be used by entities other than the client to identify this as a potential stateless reset. A server that occasionally uses different connection IDs might introduce some uncertainty about this.

The Packet Number field is set to a randomized value. The server SHOULD send a packet with a short header and a type of 0x1F. This produces the shortest possible packet number encoding, which minimizes the perceived gap between the last packet that the server sent and this packet. A server MAY use a different short header type, indicating a different packet number length, but a longer

packet number encoding might allow this message to be identified as a stateless reset more easily using heuristics.

After the Packet Number, the server pads the message with an arbitrary number of octets containing random values.

Finally, the last 16 octets of the packet are set to the value of the Stateless Reset Token.

A stateless reset is not appropriate for signaling error conditions. An endpoint that wishes to communicate a fatal connection error MUST use a CONNECTION_CLOSE or APPLICATION_CLOSE frame if it has sufficient state to do so.

This stateless reset design is specific to QUIC version 1. A server that supports multiple versions of QUIC needs to generate a stateless reset that will be accepted by clients that support any version that the server might support (or might have supported prior to losing state). Designers of new versions of QUIC need to be aware of this and either reuse this design, or use a portion of the packet other than the last 16 octets for carrying data.

6.9.4.1. Detecting a Stateless Reset

A client detects a potential stateless reset when a packet with a short header either cannot be decrypted or is marked as a duplicate packet. The client then compares the last 16 octets of the packet with the Stateless Reset Token provided by the server in its transport parameters. If these values are identical, the client MUST enter the draining period and not send any further packets on this connection. If the comparison fails, the packet can be discarded.

6.9.4.2. Calculating a Stateless Reset Token

The stateless reset token MUST be difficult to guess. In order to create a Stateless Reset Token, a server could randomly generate [RFC4086] a secret for every connection that it creates. However, this presents a coordination problem when there are multiple servers in a cluster or a storage problem for a server that might lose state. Stateless reset specifically exists to handle the case where state is lost, so this approach is suboptimal.

A single static key can be used across all connections to the same endpoint by generating the proof using a second iteration of a preimage-resistant function that takes three inputs: the static key, the server’s connection ID (see Section 4.7), and an identifier for the server instance. A server could use HMAC [RFC2104] (for example, HMAC(static_key, server_id || connection_id)) or HKDF [RFC5869] (for

example, using the static key as input keying material, with server and connection identifiers as salt). The output of this function is truncated to 16 octets to produce the Stateless Reset Token for that connection.

A server that loses state can use the same method to generate a valid Stateless Reset Secret. The connection ID comes from the packet that the server receives.

This design relies on the client always sending a connection ID in its packets so that the server can use the connection ID from a packet to reset the connection. A server that uses this design cannot allow clients to use a zero-length connection ID.

Revealing the Stateless Reset Token allows any entity to terminate the connection, so a value can only be used once. This method for choosing the Stateless Reset Token means that the combination of server instance, connection ID, and static key cannot occur for another connection. A connection ID from a connection that is reset by revealing the Stateless Reset Token cannot be reused for new connections at the same server without first changing to use a different static key or server identifier.

Note that Stateless Reset messages do not have any cryptographic protection.

7. Frame Types and Formats

As described in Section 5, packets contain one or more frames. This section describes the format and semantics of the core QUIC frame types.

7.1. Variable-Length Integer Encoding

QUIC frames use a common variable-length encoding for all non-negative integer values. This encoding ensures that smaller integer values need fewer octets to encode.

The QUIC variable-length integer encoding reserves the two most significant bits of the first octet to encode the base 2 logarithm of the integer encoding length in octets. The integer value is encoded on the remaining bits, in network byte order.

This means that integers are encoded on 1, 2, 4, or 8 octets and can encode 6, 14, 30, or 62 bit values respectively. Table 4 summarizes the encoding properties.

2Bit	Length	Usable Bits	Range
00	1	6	0-63
01	2	14	0-16383
10	4	30	0-1073741823
11	8	62	0-4611686018427387903

Table 4: Summary of Integer Encodings

For example, the eight octet sequence c2 19 7c 5e ff 14 e8 8c (in hexadecimal) decodes to the decimal value 151288809941952652; the four octet sequence 9d 7f 3e 7d decodes to 494878333; the two octet sequence 7b bd decodes to 15293; and the single octet 25 decodes to 37 (as does the two octet sequence 40 25).

Error codes (Section 11.3) are described using integers, but do not use this encoding.

7.2. PADDING Frame

The PADDING frame (type=0x00) has no semantic value. PADDING frames can be used to increase the size of a packet. Padding can be used to increase an initial client packet to the minimum required size, or to provide protection against traffic analysis for protected packets.

A PADDING frame has no content. That is, a PADDING frame consists of the single octet that identifies the frame as a PADDING frame.

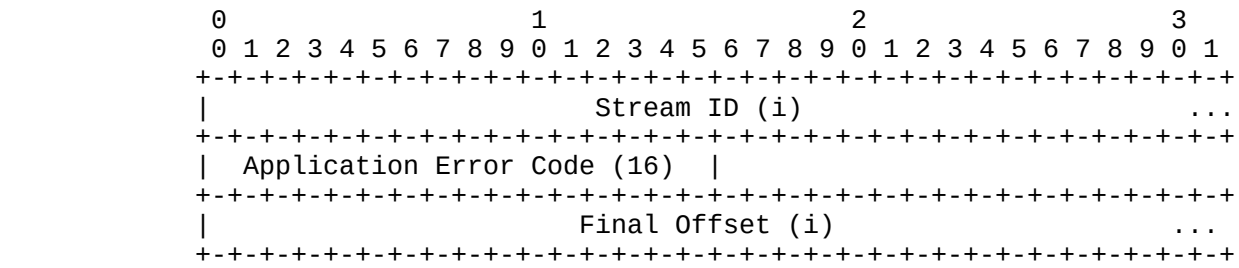
7.3. RST_STREAM Frame

An endpoint may use a RST_STREAM frame (type=0x01) to abruptly terminate a stream.

After sending a RST_STREAM, an endpoint ceases transmission and retransmission of STREAM frames on the identified stream. A receiver of RST_STREAM can discard any data that it already received on that stream.

An endpoint that receives a RST_STREAM frame for a send-only stream MUST terminate the connection with error PROTOCOL_VIOLATION.

The RST_STREAM frame is as follows:



The fields are:

Stream ID: A variable-length integer encoding of the Stream ID of the stream being terminated.

Application Protocol Error Code: A 16-bit application protocol error code (see Section 11.4) which indicates why the stream is being closed.

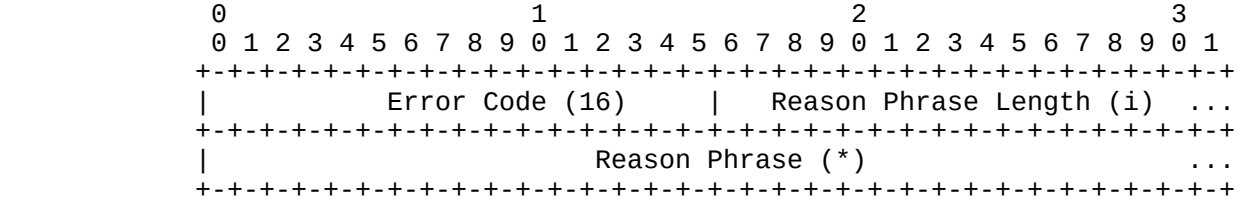
Final Offset: A variable-length integer indicating the absolute byte offset of the end of data written on this stream by the RST_STREAM sender.

7.4. CONNECTION_CLOSE frame

An endpoint sends a CONNECTION_CLOSE frame (type=0x02) to notify its peer that the connection is being closed. CONNECTION_CLOSE is used to signal errors at the QUIC layer, or the absence of errors (with the NO_ERROR code).

If there are open streams that haven't been explicitly closed, they are implicitly closed when the connection is closed.

The CONNECTION_CLOSE frame is as follows:



The fields of a CONNECTION_CLOSE frame are as follows:

Error Code: A 16-bit error code which indicates the reason for closing this connection. CONNECTION_CLOSE uses codes from the

space defined in Section 11.3 (APPLICATION_CLOSE uses codes from the application protocol error code space, see Section 11.4).

Reason Phrase Length: A variable-length integer specifying the length of the reason phrase in bytes. Note that a CONNECTION_CLOSE frame cannot be split between packets, so in practice any limits on packet size will also limit the space available for a reason phrase.

Reason Phrase: A human-readable explanation for why the connection was closed. This can be zero length if the sender chooses to not give details beyond the Error Code. This SHOULD be a UTF-8 encoded string [RFC3629].

7.5. APPLICATION_CLOSE frame

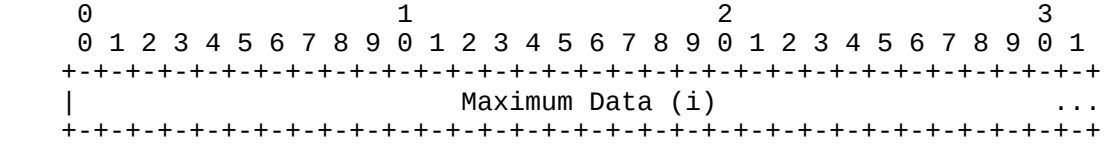
An APPLICATION_CLOSE frame (type=0x03) uses the same format as the CONNECTION_CLOSE frame (Section 7.4), except that it uses error codes from the application protocol error code space (Section 11.4) instead of the transport error code space.

Other than the error code space, the format and semantics of the APPLICATION_CLOSE frame are identical to the CONNECTION_CLOSE frame.

7.6. MAX_DATA Frame

The MAX_DATA frame (type=0x04) is used in flow control to inform the peer of the maximum amount of data that can be sent on the connection as a whole.

The frame is as follows:



The fields in the MAX_DATA frame are as follows:

Maximum Data: A variable-length integer indicating the maximum amount of data that can be sent on the entire connection, in units of octets.

All data sent in STREAM frames counts toward this limit, with the exception of data on stream 0. The sum of the largest received offsets on all streams - including streams in terminal states, but excluding stream 0 - MUST NOT exceed the value advertised by a

receiver. An endpoint MUST terminate a connection with a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA error if it receives more data than the maximum data value that it has sent, unless this is a result of a change in the initial limits (see Section 6.4.2).

7.7. MAX_STREAM_DATA Frame

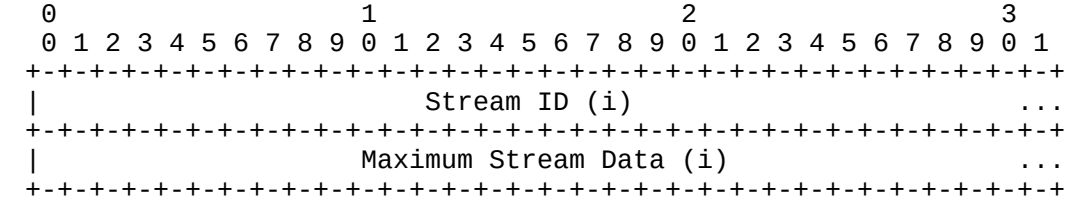
The MAX_STREAM_DATA frame (type=0x05) is used in flow control to inform a peer of the maximum amount of data that can be sent on a stream.

An endpoint that receives a MAX_STREAM_DATA frame for a receive-only stream MUST terminate the connection with error PROTOCOL_VIOLATION.

An endpoint that receives a MAX_STREAM_DATA frame for a send-only stream it has not opened MUST terminate the connection with error PROTOCOL_VIOLATION.

Note that an endpoint may legally receive a MAX_STREAM_DATA frame on a bidirectional stream it has not opened.

The frame is as follows:



The fields in the MAX_STREAM_DATA frame are as follows:

Stream ID: The stream ID of the stream that is affected encoded as a variable-length integer.

Maximum Stream Data: A variable-length integer indicating the maximum amount of data that can be sent on the identified stream, in units of octets.

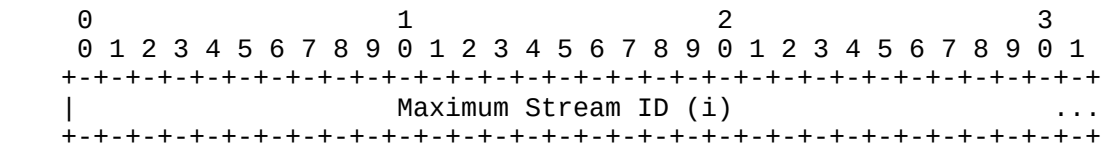
When counting data toward this limit, an endpoint accounts for the largest received offset of data that is sent or received on the stream. Loss or reordering can mean that the largest received offset on a stream can be greater than the total size of data received on that stream. Receiving STREAM frames might not increase the largest received offset.

The data sent on a stream MUST NOT exceed the largest maximum stream data value advertised by the receiver. An endpoint MUST terminate a connection with a FLOW_CONTROL_ERROR error if it receives more data than the largest maximum stream data that it has sent for the affected stream, unless this is a result of a change in the initial limits (see Section 6.4.2).

7.8. MAX_STREAM_ID Frame

The MAX_STREAM_ID frame (type=0x06) informs the peer of the maximum stream ID that they are permitted to open.

The frame is as follows:



The fields in the MAX_STREAM_ID frame are as follows:

Maximum Stream ID: ID of the maximum unidirectional or bidirectional peer-initiated stream ID for the connection encoded as a variable-length integer. The limit applies to unidirectional steams if the second least signification bit of the stream ID is 1, and applies to bidirectional streams if it is 0.

Loss or reordering can mean that a MAX_STREAM_ID frame can be received which states a lower stream limit than the client has previously received. MAX_STREAM_ID frames which do not increase the maximum stream ID MUST be ignored.

A peer MUST NOT initiate a stream with a higher stream ID than the greatest maximum stream ID it has received. An endpoint MUST terminate a connection with a STREAM_ID_ERROR error if a peer initiates a stream with a higher stream ID than it has sent, unless this is a result of a change in the initial limits (see Section 6.4.2).

7.9. PING Frame

Endpoints can use PING frames (type=0x07) to verify that their peers are still alive or to check reachability to the peer. The PING frame contains no additional fields.

The receiver of a PING frame simply needs to acknowledge the packet containing this frame.

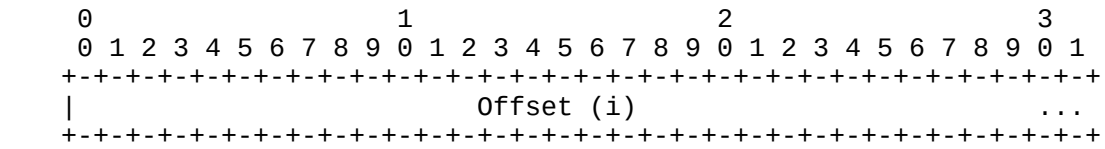
The PING frame can be used to keep a connection alive when an application or application protocol wishes to prevent the connection from timing out. An application protocol SHOULD provide guidance about the conditions under which generating a PING is recommended. This guidance SHOULD indicate whether it is the client or the server that is expected to send the PING. Having both endpoints send PING frames without coordination can produce an excessive number of packets and poor performance.

A connection will time out if no packets are sent or received for a period longer than the time specified in the idle_timeout transport parameter (see Section 6.9). However, state in middleboxes might time out earlier than that. Though REQ-5 in [RFC4787] recommends a 2 minute timeout interval, experience shows that sending packets every 15 to 30 seconds is necessary to prevent the majority of middleboxes from losing state for UDP flows.

7.10. BLOCKED Frame

A sender SHOULD send a BLOCKED frame (type=0x08) when it wishes to send data, but is unable to due to connection-level flow control (see Section 10.2.1). BLOCKED frames can be used as input to tuning of flow control algorithms (see Section 10.1.2).

The BLOCKED frame is as follows:



The BLOCKED frame contains a single field.

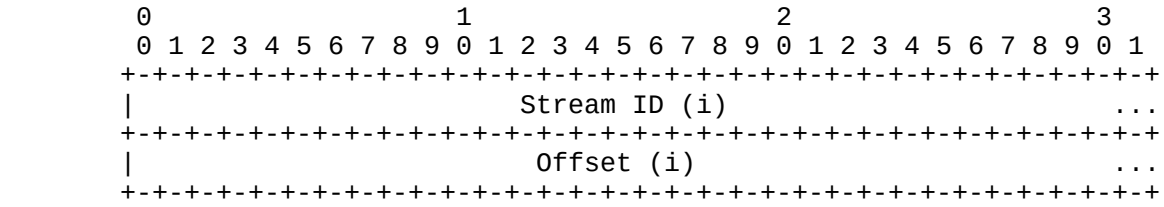
Offset: A variable-length integer indicating the connection-level offset at which the blocking occurred.

7.11. STREAM_BLOCKED Frame

A sender SHOULD send a STREAM_BLOCKED frame (type=0x09) when it wishes to send data, but is unable to due to stream-level flow control. This frame is analogous to BLOCKED (Section 7.10).

An endpoint that receives a STREAM_BLOCKED frame for a send-only stream MUST terminate the connection with error PROTOCOL_VIOLATION.

The STREAM_BLOCKED frame is as follows:



The STREAM_BLOCKED frame contains two fields:

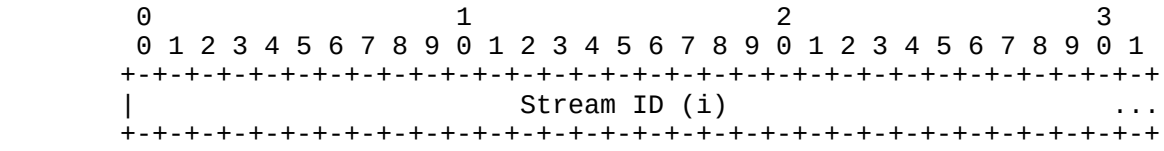
Stream ID: A variable-length integer indicating the stream which is flow control blocked.

Offset: A variable-length integer indicating the offset of the stream at which the blocking occurred.

7.12. STREAM_ID_BLOCKED Frame

A sender MAY send a STREAM_ID_BLOCKED frame (type=0x0a) when it wishes to open a stream, but is unable to due to the maximum stream ID limit set by its peer (see Section 7.8). This does not open the stream, but informs the peer that a new stream was needed, but the stream limit prevented the creation of the stream.

The STREAM_ID_BLOCKED frame is as follows:



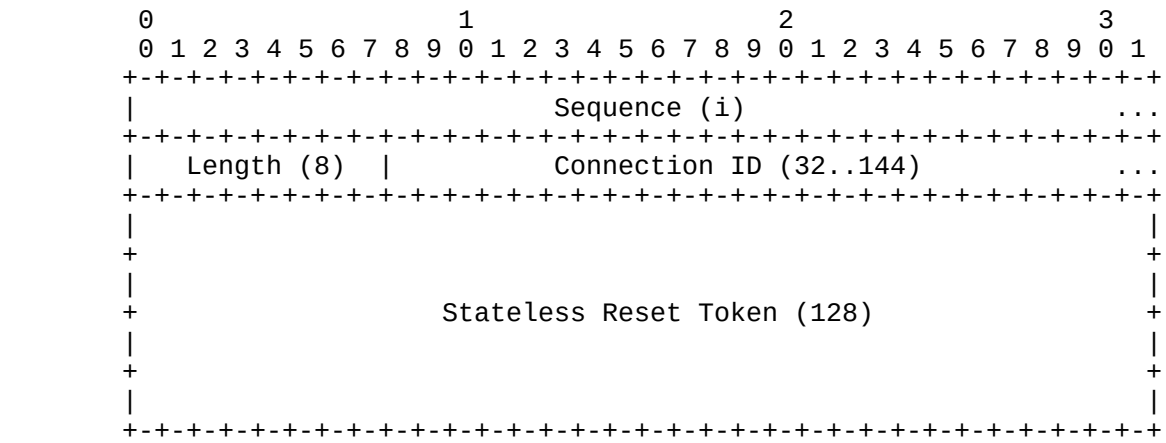
The STREAM_ID_BLOCKED frame contains a single field.

Stream ID: A variable-length integer indicating the highest stream ID that the sender was permitted to open.

7.13. NEW_CONNECTION_ID Frame

An endpoint sends a NEW_CONNECTION_ID frame (type=0x0b) to provide its peer with alternative connection IDs that can be used to break linkability when migrating connections (see Section 6.8.5).

The NEW_CONNECTION_ID is as follows:



The fields are:

Sequence: A variable-length integer. This value starts at 0 and increases by 1 for each connection ID that is provided by the server. The connection ID that is assigned during the handshake is assumed to have a sequence of -1. That is, the value selected during the handshake comes immediately before the first value that a server can send.

Length: An 8-bit unsigned integer containing the length of the connection ID. Values less than 4 and greater than 18 are invalid and MUST be treated as a connection error of type PROTOCOL_VIOLATION.

Connection ID: A connection ID of the specified length.

Stateless Reset Token: A 128-bit value that will be used to for a stateless reset when the associated connection ID is used (see Section 6.9.4).

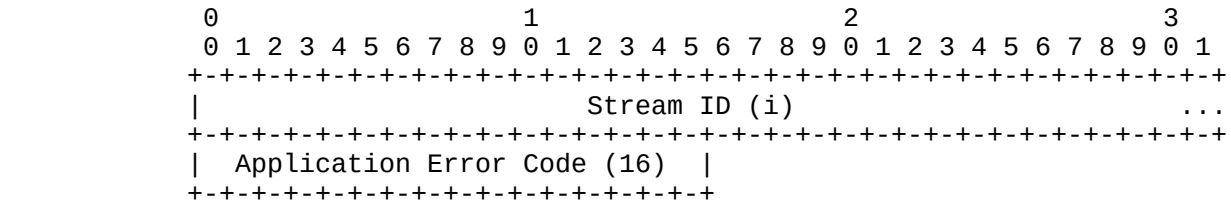
An endpoint MUST NOT send this frame if it currently requires that its peer send packets with a zero-length Destination Connection ID. Changing the length of a connection ID to or from zero-length makes it difficult to identify when the value of the connection ID changed. An endpoint that is sending packets with a zero-length Destination Connection ID MUST treat receipt of a NEW_CONNECTION_ID frame as a connection error of type PROTOCOL_VIOLATION.

7.14. STOP_SENDING Frame

An endpoint may use a STOP_SENDING frame (type=0x0c) to communicate that incoming data is being discarded on receipt at application request. This signals a peer to abruptly terminate transmission on a stream.

Receipt of a STOP_SENDING frame is only valid for a send stream that exists and is not in the "Ready" state (see Section 9.2.1). Receiving a STOP_SENDING frame for a send stream that is "Ready" or non-existent MUST be treated as a connection error of type PROTOCOL_VIOLATION. An endpoint that receives a STOP_SENDING frame for a receive-only stream MUST terminate the connection with error PROTOCOL_VIOLATION.

The STOP_SENDING frame is as follows:



The fields are:

Stream ID: A variable-length integer carrying the Stream ID of the stream being ignored.

Application Error Code: A 16-bit, application-specified reason the sender is ignoring the stream (see Section 11.4).

7.15. ACK Frame

Receivers send ACK frames (type=0x0d) to inform senders which packets they have received and processed. A sent packet that has never been acknowledged is missing. The ACK frame contains any number of ACK blocks. ACK blocks are ranges of acknowledged packets.

Unlike TCP SACKs, QUIC acknowledgements are irrevocable. Once a packet has been acknowledged, even if it does not appear in a future ACK frame, it remains acknowledged.

A client MUST NOT acknowledge Retry packets. Retry packets include the packet number from the Initial packet it responds to. Version Negotiation packets cannot be acknowledged because they do not contain a packet number. Rather than relying on ACK frames, these

packets are implicitly acknowledged by the next Initial packet sent by the client.

An ACK frame is shown below.

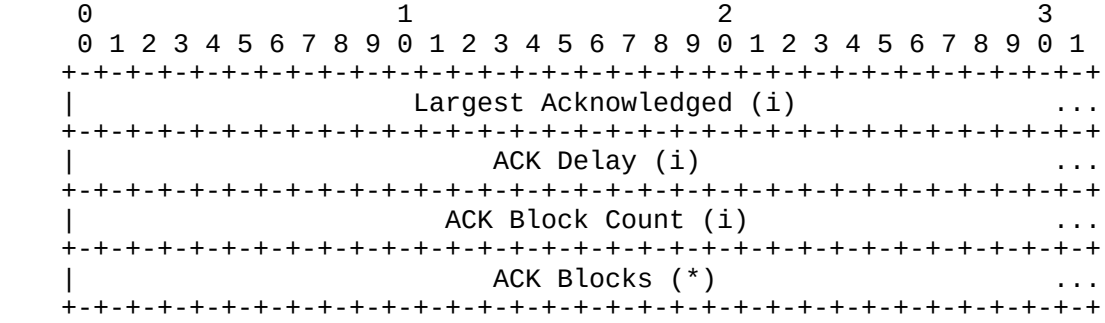


Figure 7: ACK Frame Format

The fields in the ACK frame are as follows:

Largest Acknowledged: A variable-length integer representing the largest packet number the peer is acknowledging; this is usually the largest packet number that the peer has received prior to generating the ACK frame. Unlike the packet number in the QUIC long or short header, the value in an ACK frame is not truncated.

ACK Delay: A variable-length integer including the time in microseconds that the largest acknowledged packet, as indicated in the Largest Acknowledged field, was received by this peer to when this ACK was sent. The value of the ACK Delay field is scaled by multiplying the encoded value by the 2 to the power of the value of the "ack_delay_exponent" transport parameter set by the sender of the ACK frame. The "ack_delay_exponent" defaults to 3, or a multiplier of 8 (see Section 6.4.1). Scaling in this fashion allows for a larger range of values with a shorter encoding at the cost of lower resolution.

ACK Block Count: The number of Additional ACK Block (and Gap) fields after the First ACK Block.

ACK Blocks: Contains one or more blocks of packet numbers which have been successfully received, see Section 7.15.1.

7.15.1. ACK Block Section

The ACK Block Section consists of alternating Gap and ACK Block fields in descending packet number order. A First Ack Block field is followed by a variable number of alternating Gap and Additional ACK Blocks. The number of Gap and Additional ACK Block fields is determined by the ACK Block Count field.

Gap and ACK Block fields use a relative integer encoding for efficiency. Though each encoded value is positive, the values are subtracted, so that each ACK Block describes progressively lower-numbered packets. As long as contiguous ranges of packets are small, the variable-length integer encoding ensures that each range can be expressed in a small number of octets.

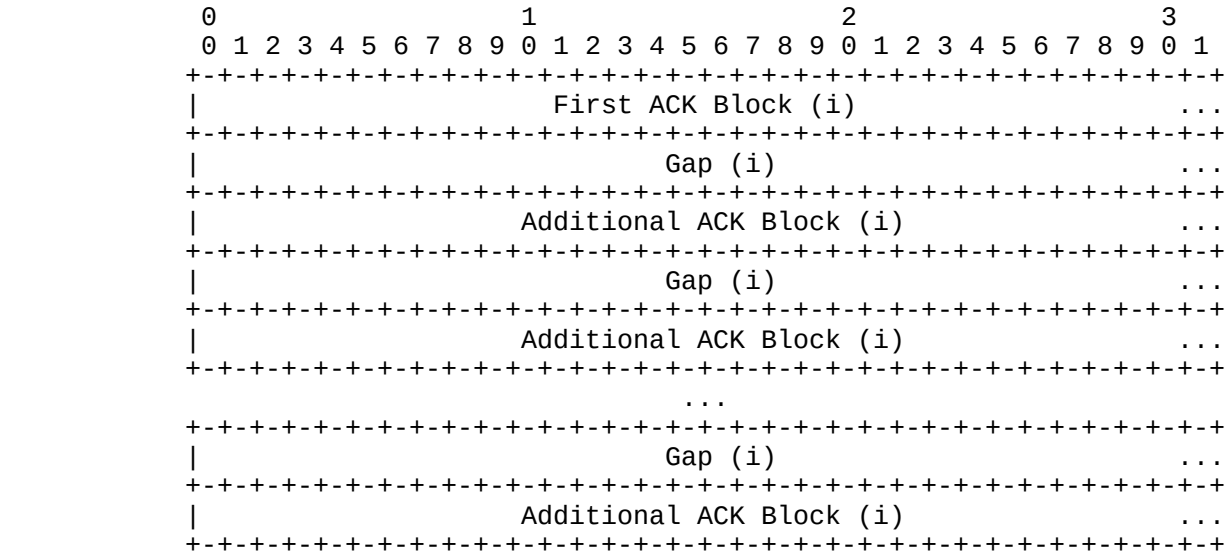


Figure 8: ACK Block Section

Each ACK Block acknowledges a contiguous range of packets by indicating the number of acknowledged packets that precede the largest packet number in that block. A value of zero indicates that only the largest packet number is acknowledged. Larger ACK Block values indicate a larger range, with corresponding lower values for the smallest packet number in the range. Thus, given a largest packet number for the ACK, the smallest value is determined by the formula:

```
smallest = largest - ack_block
```

The range of packets that are acknowledged by the ACK block include the range from the smallest packet number to the largest, inclusive.

The largest value for the First ACK Block is determined by the Largest Acknowledged field; the largest for Additional ACK Blocks is determined by cumulatively subtracting the size of all preceding ACK Blocks and Gaps.

Each Gap indicates a range of packets that are not being acknowledged. The number of packets in the gap is one higher than the encoded value of the Gap Field.

The value of the Gap field establishes the largest packet number value for the ACK block that follows the gap using the following formula:

```
largest = previous_smallest - gap - 2
```

If the calculated value for largest or smallest packet number for any ACK Block is negative, an endpoint MUST generate a connection error of type FRAME_ERROR indicating an error in an ACK frame (that is, 0x10d).

The fields in the ACK Block Section are:

First ACK Block: A variable-length integer indicating the number of contiguous packets preceding the Largest Acknowledged that are being acknowledged.

Gap (repeated): A variable-length integer indicating the number of contiguous unacknowledged packets preceding the packet number one lower than the smallest in the preceding ACK Block.

ACK Block (repeated): A variable-length integer indicating the number of contiguous acknowledged packets preceding the largest packet number, as determined by the preceding Gap.

7.15.2. Sending ACK Frames

Implementations MUST NOT generate packets that only contain ACK frames in response to packets which only contain ACK frames. However, they MUST acknowledge packets containing only ACK frames when sending ACK frames in response to other packets. Implementations MUST NOT send more than one packet containing only ACK frames per received packet that contains frames other than ACK frames. Packets containing non-ACK frames MUST be acknowledged immediately or when a delayed ack timer expires.

To limit ACK blocks to those that have not yet been received by the sender, the receiver SHOULD track which ACK frames have been acknowledged by its peer. Once an ACK frame has been acknowledged, the packets it acknowledges SHOULD NOT be acknowledged again.

Because ACK frames are not sent in response to ACK-only packets, a receiver that is only sending ACK frames will only receive acknowledgements for its packets if the sender includes them in packets with non-ACK frames. A sender SHOULD bundle ACK frames with other frames when possible.

To limit receiver state or the size of ACK frames, a receiver MAY limit the number of ACK blocks it sends. A receiver can do this even without receiving acknowledgment of its ACK frames, with the knowledge this could cause the sender to unnecessarily retransmit some data. Standard QUIC [QUIC-RECOVERY] algorithms declare packets lost after sufficiently newer packets are acknowledged. Therefore, the receiver SHOULD repeatedly acknowledge newly received packets in preference to packets received in the past.

7.15.3. ACK Frames and Packet Protection

ACK frames that acknowledge protected packets MUST be carried in a packet that has an equivalent or greater level of packet protection.

Packets that are protected with 1-RTT keys MUST be acknowledged in packets that are also protected with 1-RTT keys.

A packet that is not protected and claims to acknowledge a packet number that was sent with packet protection is not valid. An unprotected packet that carries acknowledgments for protected packets MUST be discarded in its entirety.

Packets that a client sends with 0-RTT packet protection MUST be acknowledged by the server in packets protected by 1-RTT keys. This can mean that the client is unable to use these acknowledgments if the server cryptographic handshake messages are delayed or lost. Note that the same limitation applies to other data sent by the server protected by the 1-RTT keys.

Unprotected packets, such as those that carry the initial cryptographic handshake messages, MAY be acknowledged in unprotected packets. Unprotected packets are vulnerable to falsification or modification. Unprotected packets can be acknowledged along with protected packets in a protected packet.

An endpoint SHOULD acknowledge packets containing cryptographic handshake messages in the next unprotected packet that it sends,

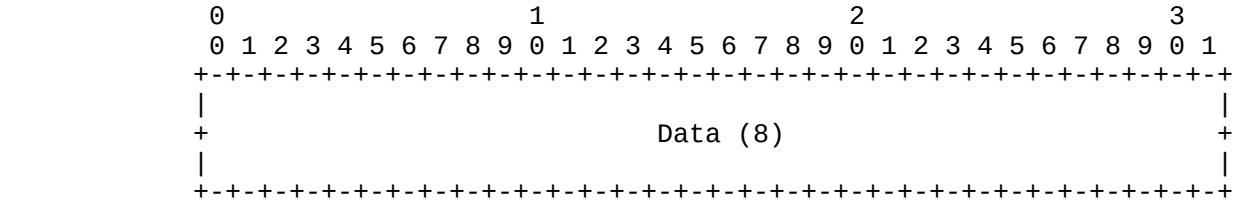
unless it is able to acknowledge those packets in later packets protected by 1-RTT keys. At the completion of the cryptographic handshake, both peers send unprotected packets containing cryptographic handshake messages followed by packets protected by 1-RTT keys. An endpoint SHOULD acknowledge the unprotected packets that complete the cryptographic handshake in a protected packet, because its peer is guaranteed to have access to 1-RTT packet protection keys.

For instance, a server acknowledges a TLS ClientHello in the packet that carries the TLS ServerHello; similarly, a client can acknowledge a TLS HelloRetryRequest in the packet containing a second TLS ClientHello. The complete set of server handshake messages (TLS ServerHello through to Finished) might be acknowledged by a client in protected packets, because it is certain that the server is able to decipher the packet.

7.16. PATH_CHALLENGE Frame

Endpoints can use PATH_CHALLENGE frames (type=0x0e) to check reachability to the peer and for path validation during connection establishment and connection migration.

PATH_CHALLENGE frames contain an 8-byte payload.



Data: This 8-byte field contains arbitrary data.

A PATH_CHALLENGE frame containing 8 octets that are hard to guess is sufficient to ensure that it is easier to receive the packet than it is to guess the value correctly.

The recipient of this frame MUST generate a PATH_RESPONSE frame (Section 7.17) containing the same Data.

7.17. PATH_RESPONSE Frame

The PATH_RESPONSE frame (type=0x0f) is sent in response to a PATH_CHALLENGE frame. Its format is identical to the PATH_CHALLENGE frame (Section 7.16).

If the content of a PATH_RESPONSE frame does not match the content of a PATH_CHALLENGE frame previously sent by the endpoint, the endpoint MAY generate a connection error of type UNSOLICITED_PATH_RESPONSE.

7.18. STREAM Frames

STREAM frames implicitly create a stream and carry stream data. The STREAM frame takes the form 0b00010XXX (or the set of values from 0x10 to 0x17). The value of the three low-order bits of the frame type determine the fields that are present in the frame.

- The OFF bit (0x04) in the frame type is set to indicate that there is an Offset field present. When set to 1, the Offset field is present; when set to 0, the Offset field is absent and the Stream Data starts at an offset of 0 (that is, the frame contains the first octets of the stream, or the end of a stream that includes no data).

- The LEN bit (0x02) in the frame type is set to indicate that there is a Length field present. If this bit is set to 0, the Length field is absent and the Stream Data field extends to the end of the packet. If this bit is set to 1, the Length field is present.

- The FIN bit (0x01) of the frame type is set only on frames that contain the final offset of the stream. Setting this bit indicates that the frame marks the end of the stream.

An endpoint that receives a STREAM frame for a send-only stream MUST terminate the connection with error PROTOCOL_VIOLATION.

A STREAM frame is shown below.

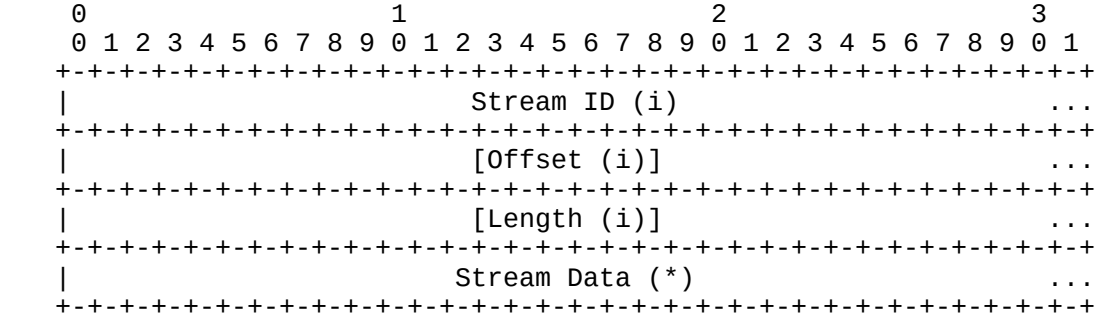


Figure 9: STREAM Frame Format

The STREAM frame contains the following fields:

Stream ID: A variable-length integer indicating the stream ID of the stream (see Section 9.1).

Offset: A variable-length integer specifying the byte offset in the stream for the data in this STREAM frame. This field is present when the OFF bit is set to 1. When the Offset field is absent, the offset is 0.

Length: A variable-length integer specifying the length of the Stream Data field in this STREAM frame. This field is present when the LEN bit is set to 1. When the LEN bit is set to 0, the Stream Data field consumes all the remaining octets in the packet.

Stream Data: The bytes from the designated stream to be delivered.

A stream frame's Stream Data MUST NOT be empty, unless the offset is 0 or the FIN bit is set. When the FIN flag is sent on an empty STREAM frame, the offset in the STREAM frame is the offset of the next byte that would be sent.

The first byte in the stream has an offset of 0. The largest offset delivered on a stream - the sum of the re-constructed offset and data length - MUST be less than 2^62.

Stream multiplexing is achieved by interleaving STREAM frames from multiple streams into one or more QUIC packets. A single QUIC packet can include multiple STREAM frames from one or more streams.

Implementation note: One of the benefits of QUIC is avoidance of head-of-line blocking across multiple streams. When a packet loss occurs, only streams with data in that packet are blocked waiting for a retransmission to be received, while other streams can continue making progress. Note that when data from multiple streams is bundled into a single QUIC packet, loss of that packet blocks all those streams from making progress. An implementation is therefore advised to bundle as few streams as necessary in outgoing packets without losing transmission efficiency to underfilled packets.

8. Packetization and Reliability

A sender bundles one or more frames in a QUIC packet (see Section 5).

A sender SHOULD minimize per-packet bandwidth and computational costs by bundling as many frames as possible within a QUIC packet. A sender MAY wait for a short period of time to bundle multiple frames before sending a packet that is not maximally packed, to avoid sending out large numbers of small packets. An implementation may use knowledge about application sending behavior or heuristics to

determine whether and for how long to wait. This waiting period is an implementation decision, and an implementation should be careful to delay conservatively, since any delay is likely to increase application-visible latency.

8.1. Packet Processing and Acknowledgment

A packet MUST NOT be acknowledged until packet protection has been successfully removed and all frames contained in the packet have been processed. Any stream state transitions triggered by the frame MUST have occurred. For STREAM frames, this means the data has been enqueued in preparation to be received by the application protocol, but it does not require that data is delivered and consumed.

Once the packet has been fully processed, a receiver acknowledges receipt by sending one or more ACK frames containing the packet number of the received packet. To avoid creating an indefinite feedback loop, an endpoint MUST NOT send an ACK frame in response to a packet containing only ACK or PADDING frames, even if there are packet gaps which precede the received packet. The endpoint MUST acknowledge packets containing only ACK or PADDING frames in the next ACK frame that it sends.

Strategies and implications of the frequency of generating acknowledgments are discussed in more detail in [QUIC-RECOVERY].

8.2. Retransmission of Information

QUIC packets that are determined to be lost are not retransmitted whole. The same applies to the frames that are contained within lost packets. Instead, the information that might be carried in frames is sent again in new frames as needed.

New frames and packets are used to carry information that is determined to have been lost. In general, information is sent again when a packet containing that information is determined to be lost and sending ceases when a packet containing that information is acknowledged.

- Application data sent in STREAM frames is retransmitted in new STREAM frames unless the endpoint has sent a RST_STREAM for that stream. Once an endpoint sends a RST_STREAM frame, no further STREAM frames are needed.

- The most recent set of acknowledgments are sent in ACK frames. An ACK frame SHOULD contain all unacknowledged acknowledgments, as described in Section 7.15.2.

- Cancellation of stream transmission, as carried in a RST_STREAM frame, is sent until acknowledged or until all stream data is acknowledged by the peer (that is, either the "Reset Recvd" or "Data Recvd" state is reached on the send stream). The content of a RST_STREAM frame MUST NOT change when it is sent again.

- Similarly, a request to cancel stream transmission, as encoded in a STOP_SENDING frame, is sent until the receive stream enters either a "Data Recvd" or "Reset Recvd" state, see Section 9.3.

- Connection close signals, including those that use CONNECTION_CLOSE and APPLICATION_CLOSE frames, are not sent again when packet loss is detected, but as described in Section 6.9.

- The current connection maximum data is sent in MAX_DATA frames. An updated value is sent in a MAX_DATA frame if the packet containing the most recently sent MAX_DATA frame is declared lost, or when the endpoint decides to update the limit. Care is necessary to avoid sending this frame too often as the limit can increase frequently and cause an unnecessarily large number of MAX_DATA frames to be sent.

- The current maximum stream data offset is sent in MAX_STREAM_DATA frames. Like MAX_DATA, an updated value is sent when the packet containing the most recent MAX_STREAM_DATA frame for a stream is lost or when the limit is updated, with care taken to prevent the frame from being sent too often. An endpoint SHOULD stop sending MAX_STREAM_DATA frames when the receive stream enters a "Size Known" state.

- The maximum stream ID for a stream of a given type is sent in MAX_STREAM_ID frames. Like MAX_DATA, an updated value is sent when a packet containing the most recent MAX_STREAM_ID for a stream type frame is declared lost or when the limit is updated, with care taken to prevent the frame from being sent too often.

- Blocked signals are carried in BLOCKED, STREAM_BLOCKED, and STREAM_ID_BLOCKED frames. BLOCKED streams have connection scope, STREAM_BLOCKED frames have stream scope, and STREAM_ID_BLOCKED frames are scoped to a specific stream type. New frames are sent if packets containing the most recent frame for a scope is lost, but only while the endpoint is blocked on the corresponding limit. These frames always include the limit that is causing blocking at the time that they are transmitted.

- A liveness or path validation check using PATH_CHALLENGE frames is sent periodically until a matching PATH_RESPONSE frame is received or until there is no remaining need for liveness or path

validation checking. PATH_CHALLENGE frames include a different payload each time they are sent.

- Responses to path validation using PATH_RESPONSE frames are sent just once. A new PATH_CHALLENGE frame will be sent if another PATH_RESPONSE frame is needed.

- New connection IDs are sent in NEW_CONNECTION_ID frames and retransmitted if the packet containing them is lost.

- PADDING frames contain no information, so lost PADDING frames do not require repair.

Upon detecting losses, a sender MUST take appropriate congestion control action. The details of loss detection and congestion control are described in [QUIC-RECOVERY].

8.3. Packet Size

The QUIC packet size includes the QUIC header and integrity check, but not the UDP or IP header.

Clients MUST pad any Initial packet it sends to have a QUIC packet size of at least 1200 octets. Sending an Initial packet of this size ensures that the network path supports a reasonably sized packet, and helps reduce the amplitude of amplification attacks caused by server responses toward an unverified client address.

An Initial packet MAY exceed 1200 octets if the client knows that the Path Maximum Transmission Unit (PMTU) supports the size that it chooses.

A server MAY send a CONNECTION_CLOSE frame with error code PROTOCOL_VIOLATION in response to an Initial packet smaller than 1200 octets. It MUST NOT send any other frame type in response, or otherwise behave as if any part of the offending packet was processed as valid.

8.4. Path Maximum Transmission Unit

The Path Maximum Transmission Unit (PMTU) is the maximum size of the entire IP header, UDP header, and UDP payload. The UDP payload includes the QUIC packet header, protected payload, and any authentication fields.

All QUIC packets SHOULD be sized to fit within the estimated PMTU to avoid IP fragmentation or packet drops. To optimize bandwidth efficiency, endpoints SHOULD use Packetization Layer PMTU Discovery

([PLPMTUD]). Endpoints MAY use PMTU Discovery ([PMTUDv4], [PMTUDv6]) for detecting the PMTU, setting the PMTU appropriately, and storing the result of previous PMTU determinations.

In the absence of these mechanisms, QUIC endpoints SHOULD NOT send IP packets larger than 1280 octets. Assuming the minimum IP header size, this results in a QUIC packet size of 1232 octets for IPv6 and 1252 octets for IPv4. Some QUIC implementations MAY wish to be more conservative in computing allowed QUIC packet size given unknown tunneling overheads or IP header options.

QUIC endpoints that implement any kind of PMTU discovery SHOULD maintain an estimate for each combination of local and remote IP addresses. Each pairing of local and remote addresses could have a different maximum MTU in the path.

QUIC depends on the network path supporting a MTU of at least 1280 octets. This is the IPv6 minimum MTU and therefore also supported by most modern IPv4 networks. An endpoint MUST NOT reduce its MTU below this number, even if it receives signals that indicate a smaller limit might exist.

If a QUIC endpoint determines that the PMTU between any pair of local and remote IP addresses has fallen below 1280 octets, it MUST immediately cease sending QUIC packets on the affected path. This could result in termination of the connection if an alternative path cannot be found.

8.4.1. Special Considerations for PMTU Discovery

Traditional ICMP-based path MTU discovery in IPv4 [RFC1191] is potentially vulnerable to off-path attacks that successfully guess the IP/port 4-tuple and reduce the MTU to a bandwidth-inefficient value. TCP connections mitigate this risk by using the (at minimum) 8 bytes of transport header echoed in the ICMP message to validate the TCP sequence number as valid for the current connection. However, as QUIC operates over UDP, in IPv4 the echoed information could consist only of the IP and UDP headers, which usually has insufficient entropy to mitigate off-path attacks.

As a result, endpoints that implement PMTUD in IPv4 SHOULD take steps to mitigate this risk. For instance, an application could:

- Set the IPv4 Don’t Fragment (DF) bit on a small proportion of packets, so that most invalid ICMP messages arrive when there are no DF packets outstanding, and can therefore be identified as spurious.

- Store additional information from the IP or UDP headers from DF packets (for example, the IP ID or UDP checksum) to further authenticate incoming Datagram Too Big messages.

- Any reduction in PMTU due to a report contained in an ICMP packet is provisional until QUIC's loss detection algorithm determines that the packet is actually lost.

8.4.2. Special Considerations for Packetization Layer PMTU Discovery

The PADDING frame provides a useful option for PMTU probe packets that does not exist in other transports. PADDING frames generate acknowledgements, but their content need not be delivered reliably. PADDING frames may delay the delivery of application data, as they consume the congestion window. However, by definition their likely loss in a probe packet does not require delay-inducing retransmission of application data.

When implementing the algorithm in Section 7.2 of [RFC4821], the initial value of search_low SHOULD be consistent with the IPv6 minimum packet size. Paths that do not support this size cannot deliver Initial packets, and therefore are not QUIC-compliant.

Section 7.3 of [RFC4821] discusses tradeoffs between small and large increases in the size of probe packets. As QUIC probe packets need not contain application data, aggressive increases in probe size carry fewer consequences.

9. Streams: QUIC's Data Structuring Abstraction

Streams in QUIC provide a lightweight, ordered byte-stream abstraction.

There are two basic types of stream in QUIC. Unidirectional streams carry data in one direction only; bidirectional streams allow for data to be sent in both directions. Different stream identifiers are used to distinguish between unidirectional and bidirectional streams, as well as to create a separation between streams that are initiated by the client and server (see Section 9.1).

Either type of stream can be created by either endpoint, can concurrently send data interleaved with other streams, and can be cancelled.

Stream offsets allow for the octets on a stream to be placed in order. An endpoint MUST be capable of delivering data received on a stream in order. Implementations MAY choose to offer the ability to

deliver data out of order. There is no means of ensuring ordering between octets on different streams.

The creation and destruction of streams are expected to have minimal bandwidth and computational cost. A single STREAM frame may create, carry data for, and terminate a stream, or a stream may last the entire duration of a connection.

Streams are individually flow controlled, allowing an endpoint to limit memory commitment and to apply back pressure. The creation of streams is also flow controlled, with each peer declaring the maximum stream ID it is willing to accept at a given time.

An alternative view of QUIC streams is as an elastic "message" abstraction, similar to the way ephemeral streams are used in SST [SST], which may be a more appealing description for some applications.

9.1. Stream Identifiers

Streams are identified by an unsigned 62-bit integer, referred to as the Stream ID. The least significant two bits of the Stream ID are used to identify the type of stream (unidirectional or bidirectional) and the initiator of the stream.

The least significant bit (0x1) of the Stream ID identifies the initiator of the stream. Clients initiate even-numbered streams (those with the least significant bit set to 0); servers initiate odd-numbered streams (with the bit set to 1). Separation of the stream identifiers ensures that client and server are able to open streams without the latency imposed by negotiating for an identifier.

If an endpoint receives a frame for a stream that it expects to initiate (i.e., odd-numbered for the client or even-numbered for the server), but which it has not yet opened, it MUST close the connection with error code STREAM_STATE_ERROR.

The second least significant bit (0x2) of the Stream ID differentiates between unidirectional streams and bidirectional streams. Unidirectional streams always have this bit set to 1 and bidirectional streams have this bit set to 0.

The two type bits from a Stream ID therefore identify streams as summarized in Table 5.

Low Bits	Stream Type
0x0	Client-Initiated, Bidirectional
0x1	Server-Initiated, Bidirectional
0x2	Client-Initiated, Unidirectional
0x3	Server-Initiated, Unidirectional

Table 5: Stream ID Types

Stream ID 0 (0x0) is a client-initiated, bidirectional stream that is used for the cryptographic handshake. Stream 0 MUST NOT be used for application data.

A QUIC endpoint MUST NOT reuse a Stream ID. Streams can be used in any order. Streams that are used out of order result in opening all lower-numbered streams of the same type in the same direction.

Stream IDs are encoded as a variable-length integer (see Section 7.1).

9.2. Stream States

This section describes the two types of QUIC stream in terms of the states of their send or receive components. Two state machines are described: one for streams on which an endpoint transmits data (Section 9.2.1); another for streams from which an endpoint receives data (Section 9.2.2).

Unidirectional streams use the applicable state machine directly. Bidirectional streams use both state machines. For the most part, the use of these state machines is the same whether the stream is unidirectional or bidirectional. The conditions for opening a stream are slightly more complex for a bidirectional stream because the opening of either send or receive sides causes the stream to open in both directions.

An endpoint can open streams up to its maximum stream limit in any order, however endpoints SHOULD open the send side of streams for each type in order.

Note: These states are largely informative. This document uses stream states to describe rules for when and how different types of frames can be sent and the reactions that are expected when

different types of frames are received. Though these state machines are intended to be useful in implementing QUIC, these states aren't intended to constrain implementations. An implementation can define a different state machine as long as its behavior is consistent with an implementation that implements these states.

9.2.1. Send Stream States

Figure 10 shows the states for the part of a stream that sends data to a peer.

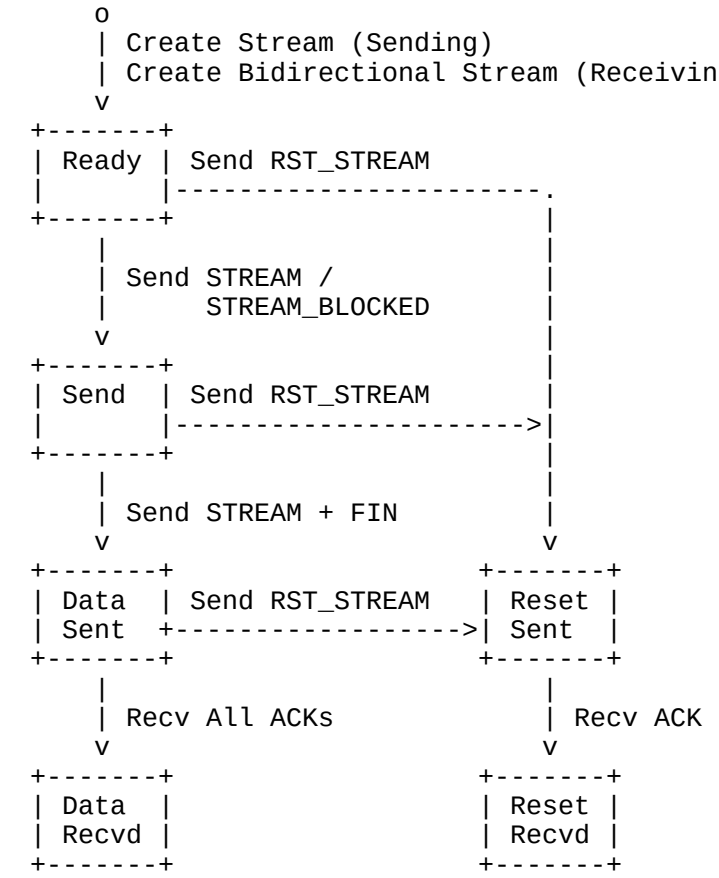


Figure 10: States for Send Streams

The sending part of stream that the endpoint initiates (types 0 and 2 for clients, 1 and 3 for servers) is opened by the application or application protocol. The "Ready" state represents a newly created

stream that is able to accept data from the application. Stream data might be buffered in this state in preparation for sending.

The sending part of a bidirectional stream initiated by a peer (type 0 for a server, type 1 for a client) enters the "Ready" state if the receiving part enters the "Recv" state.

Sending the first STREAM or STREAM_BLOCKED frame causes a send stream to enter the "Send" state. An implementation might choose to defer allocating a Stream ID to a send stream until it sends the first frame and enters this state, which can allow for better stream prioritization.

In the "Send" state, an endpoint transmits - and retransmits as necessary - data in STREAM frames. The endpoint respects the flow control limits of its peer, accepting MAX_STREAM_DATA frames. An endpoint in the "Send" state generates STREAM_BLOCKED frames if it encounters flow control limits.

After the application indicates that stream data is complete and a STREAM frame containing the FIN bit is sent, the send stream enters the "Data Sent" state. From this state, the endpoint only retransmits stream data as necessary. The endpoint no longer needs to track flow control limits or send STREAM_BLOCKED frames for a send stream in this state. The endpoint can ignore any MAX_STREAM_DATA frames it receives from its peer in this state; MAX_STREAM_DATA frames might be received until the peer receives the final stream offset.

Once all stream data has been successfully acknowledged, the send stream enters the "Data Recvd" state, which is a terminal state.

From any of the "Ready", "Send", or "Data Sent" states, an application can signal that it wishes to abandon transmission of stream data. Similarly, the endpoint might receive a STOP_SENDING frame from its peer. In either case, the endpoint sends a RST_STREAM frame, which causes the stream to enter the "Reset Sent" state.

An endpoint MAY send a RST_STREAM as the first frame on a send stream; this causes the send stream to open and then immediately transition to the "Reset Sent" state.

Once a packet containing a RST_STREAM has been acknowledged, the send stream enters the "Reset Recvd" state, which is a terminal state.

9.2.2. Receive Stream States

Figure 11 shows the states for the part of a stream that receives data from a peer. The states for a receive stream mirror only some of the states of the send stream at the peer. A receive stream doesn't track states on the send stream that cannot be observed, such as the "Ready" state; instead, receive streams track the delivery of data to the application or application protocol some of which cannot be observed by the sender.

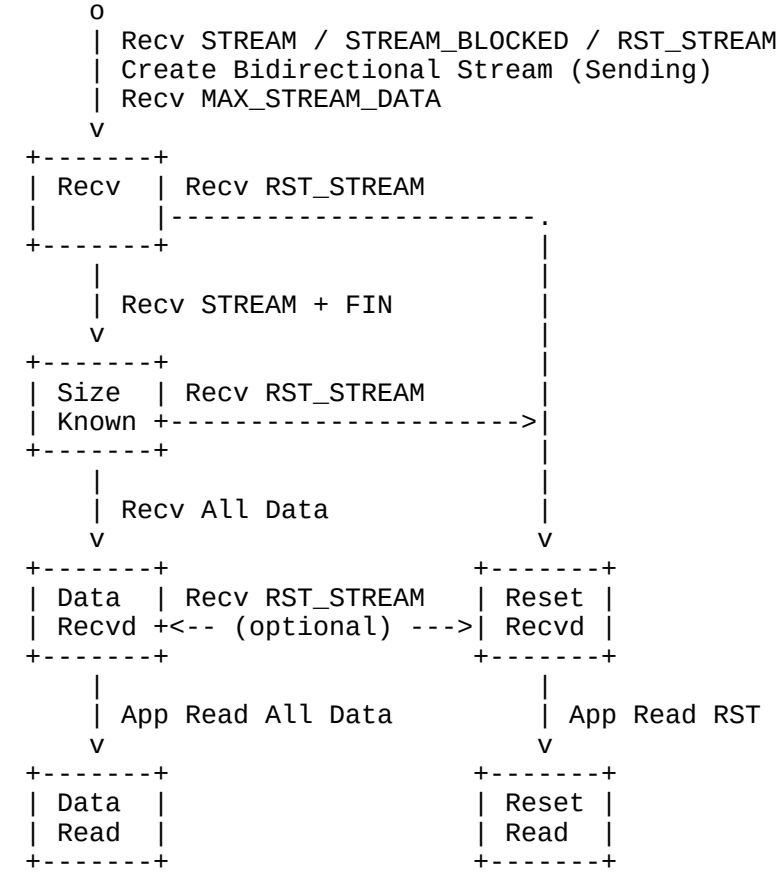


Figure 11: States for Receive Streams

The receiving part of a stream initiated by a peer (types 1 and 3 for a client, or 0 and 2 for a server) are created when the first STREAM, STREAM_BLOCKED, RST_STREAM, or MAX_STREAM_DATA (bidirectional only, see below) is received for that stream. The initial state for a

receive stream is "Recv". Receiving a RST_STREAM frame causes the receive stream to immediately transition to the "Reset Recvd".

The receive stream enters the "Recv" state when the sending part of a bidirectional stream initiated by the endpoint (type 0 for a client, type 1 for a server) enters the "Ready" state.

A bidirectional stream also opens when a MAX_STREAM_DATA frame is received. Receiving a MAX_STREAM_DATA frame implies that the remote peer has opened the stream and is providing flow control credit. A MAX_STREAM_DATA frame might arrive before a STREAM or STREAM_BLOCKED frame if packets are lost or reordered.

In the "Recv" state, the endpoint receives STREAM and STREAM_BLOCKED frames. Incoming data is buffered and can be reassembled into the correct order for delivery to the application. As data is consumed by the application and buffer space becomes available, the endpoint sends MAX_STREAM_DATA frames to allow the peer to send more data.

When a STREAM frame with a FIN bit is received, the final offset (see Section 10.3) is known. The receive stream enters the "Size Known" state. In this state, the endpoint no longer needs to send MAX_STREAM_DATA frames, it only receives any retransmissions of stream data.

Once all data for the stream has been received, the receive stream enters the "Data Recvd" state. This might happen as a result of receiving the same STREAM frame that causes the transition to "Size Known". In this state, the endpoint has all stream data. Any STREAM or STREAM_BLOCKED frames it receives for the stream can be discarded.

The "Data Recvd" state persists until stream data has been delivered to the application or application protocol. Once stream data has been delivered, the stream enters the "Data Read" state, which is a terminal state.

Receiving a RST_STREAM frame in the "Recv" or "Size Known" states causes the stream to enter the "Reset Recvd" state. This might cause the delivery of stream data to the application to be interrupted.

It is possible that all stream data is received when a RST_STREAM is received (that is, from the "Data Recvd" state). Similarly, it is possible for remaining stream data to arrive after receiving a RST_STREAM frame (the "Reset Recvd" state). An implementation is able to manage this situation as they choose. Sending RST_STREAM means that an endpoint cannot guarantee delivery of stream data; however there is no requirement that stream data not be delivered if a RST_STREAM is received. An implementation MAY interrupt delivery

of stream data, discard any data that was not consumed, and signal the existence of the RST_STREAM immediately. Alternatively, the RST_STREAM signal might be suppressed or withheld if stream data is completely received. In the latter case, the receive stream effectively transitions to "Data Recvd" from "Reset Recvd".

Once the application has been delivered the signal indicating that the receive stream was reset, the receive stream transitions to the "Reset Read" state, which is a terminal state.

9.2.3. Permitted Frame Types

The sender of a stream sends just three frame types that affect the state of a stream at either sender or receiver: STREAM (Section 7.18), STREAM_BLOCKED (Section 7.11), and RST_STREAM (Section 7.3).

A sender MUST NOT send any of these frames from a terminal state ("Data Recvd" or "Reset Recvd"). A sender MUST NOT send STREAM or STREAM_BLOCKED after sending a RST_STREAM; that is, in the "Reset Sent" state in addition to the terminal states. A receiver could receive any of these frames in any state, but only due to the possibility of delayed delivery of packets carrying them.

The receiver of a stream sends MAX_STREAM_DATA (Section 7.7) and STOP_SENDING frames (Section 7.14).

The receiver only sends MAX_STREAM_DATA in the "Recv" state. A receiver can send STOP_SENDING in any state where it has not received a RST_STREAM frame; that is states other than "Reset Recvd" or "Reset Read". However there is little value in sending a STOP_SENDING frame after all stream data has been received in the "Data Recvd" state. A sender could receive these frames in any state as a result of delayed delivery of packets.

9.2.4. Bidirectional Stream States

A bidirectional stream is composed of a send stream and a receive stream. Implementations may represent states of the bidirectional stream as composites of send and receive stream states. The simplest model presents the stream as "open" when either send or receive stream is in a non-terminal state and "closed" when both send and receive streams are in a terminal state.

Table 6 shows a more complex mapping of bidirectional stream states that loosely correspond to the stream states in HTTP/2 [HTTP2]. This shows that multiple states on send or receive streams are mapped to the same composite state. Note that this is just one possibility for

such a mapping; this mapping requires that data is acknowledged before the transition to a "closed" or "half-closed" state.

Send Stream	Receive Stream	Composite State
No Stream/Ready	No Stream/Recv *1	idle
Ready/Send/Data Sent	Recv/Size Known	open
Ready/Send/Data Sent	Data Recvd/Data Read	half-closed (remote)
Ready/Send/Data Sent	Reset Recvd/Reset Read	half-closed (remote)
Data Recvd	Recv/Size Known	half-closed (local)
Reset Sent/Reset Recvd	Recv/Size Known	half-closed (local)
Data Recvd	Recv/Size Known	half-closed (local)
Reset Sent/Reset Recvd	Data Recvd/Data Read	closed
Reset Sent/Reset Recvd	Reset Recvd/Reset Read	closed
Data Recvd	Data Recvd/Data Read	closed
Data Recvd	Reset Recvd/Reset Read	closed

Table 6: Possible Mapping of Stream States to HTTP/2

Note (*1): A stream is considered "idle" if it has not yet been created, or if the receive stream is in the "Recv" state without yet having received any frames.

9.3. Solicited State Transitions

If an endpoint is no longer interested in the data it is receiving on a stream, it MAY send a STOP_SENDING frame identifying that stream to prompt closure of the stream in the opposite direction. This typically indicates that the receiving application is no longer

reading data it receives from the stream, but is not a guarantee that incoming data will be ignored.

STREAM frames received after sending STOP_SENDING are still counted toward the connection and stream flow-control windows, even though these frames will be discarded upon receipt. This avoids potential ambiguity about which STREAM frames count toward flow control.

A STOP_SENDING frame requests that the receiving endpoint send a RST_STREAM frame. An endpoint that receives a STOP_SENDING frame MUST send a RST_STREAM frame for that stream, and can use an error code of STOPPING. If the STOP_SENDING frame is received on a send stream that is already in the "Data Sent" state, a RST_STREAM frame MAY still be sent in order to cancel retransmission of previously-sent STREAM frames.

STOP_SENDING SHOULD only be sent for a receive stream that has not been reset. STOP_SENDING is most useful for streams in the "Recv" or "Size Known" states.

An endpoint is expected to send another STOP_SENDING frame if a packet containing a previous STOP_SENDING is lost. However, once either all stream data or a RST_STREAM frame has been received for the stream - that is, the stream is in any state other than "Recv" or "Size Known" - sending a STOP_SENDING frame is unnecessary.

9.4. Stream Concurrency

An endpoint limits the number of concurrently active incoming streams by adjusting the maximum stream ID. An initial value is set in the transport parameters (see Section 6.4.1) and is subsequently increased by MAX_STREAM_ID frames (see Section 7.8).

The maximum stream ID is specific to each endpoint and applies only to the peer that receives the setting. That is, clients specify the maximum stream ID the server can initiate, and servers specify the maximum stream ID the client can initiate. Each endpoint may respond on streams initiated by the other peer, regardless of whether it is permitted to initiated new streams.

Endpoints MUST NOT exceed the limit set by their peer. An endpoint that receives a STREAM frame with an ID greater than the limit it has sent MUST treat this as a stream error of type STREAM_ID_ERROR (Section 11), unless this is a result of a change in the initial offsets (see Section 6.4.2).

A receiver MUST NOT renege on an advertisement; that is, once a receiver advertises a stream ID via a MAX_STREAM_ID frame, it MUST

NOT subsequently advertise a smaller maximum ID. A sender may receive MAX_STREAM_ID frames out of order; a sender MUST therefore ignore any MAX_STREAM_ID that does not increase the maximum.

9.5. Sending and Receiving Data

Once a stream is created, endpoints may use the stream to send and receive data. Each endpoint may send a series of STREAM frames encapsulating data on a stream until the stream is terminated in that direction. Streams are an ordered byte-stream abstraction, and they have no other structure within them. STREAM frame boundaries are not expected to be preserved in retransmissions from the sender or during delivery to the application at the receiver.

When new data is to be sent on a stream, a sender MUST set the encapsulating STREAM frame's offset field to the stream offset of the first byte of this new data. The first octet of data on a stream has an offset of 0. An endpoint is expected to send every stream octet. The largest offset delivered on a stream MUST be less than 2^62. A receiver MUST ensure that received stream data is delivered to the application as an ordered byte-stream. Data received out of order MUST be buffered for later delivery, as long as it is not in violation of the receiver's flow control limits.

An endpoint MUST NOT send data on any stream without ensuring that it is within the data limits set by its peer. The cryptographic handshake stream, Stream 0, is exempt from the connection-level data limits established by MAX_DATA. Data on stream 0 other than the initial cryptographic handshake message is still subject to stream-level data limits and MAX_STREAM_DATA. This message is exempt from flow control because it needs to be sent in a single packet regardless of the server's flow control state. This rule applies even for 0-RTT handshakes where the remembered value of MAX_STREAM_DATA would not permit sending a full initial cryptographic handshake message.

Flow control is described in detail in Section 10, and congestion control is described in the companion document [QUIC-RECOVERY].

9.6. Stream Prioritization

Stream multiplexing has a significant effect on application performance if resources allocated to streams are correctly prioritized. Experience with other multiplexed protocols, such as HTTP/2 [HTTP2], shows that effective prioritization strategies have a significant positive impact on performance.

QUIC does not provide frames for exchanging prioritization information. Instead it relies on receiving priority information from the application that uses QUIC. Protocols that use QUIC are able to define any prioritization scheme that suits their application semantics. A protocol might define explicit messages for signaling priority, such as those defined in HTTP/2; it could define rules that allow an endpoint to determine priority based on context; or it could leave the determination to the application.

A QUIC implementation SHOULD provide ways in which an application can indicate the relative priority of streams. When deciding which streams to dedicate resources to, QUIC SHOULD use the information provided by the application. Failure to account for priority of streams can result in suboptimal performance.

Stream priority is most relevant when deciding which stream data will be transmitted. Often, there will be limits on what can be transmitted as a result of connection flow control or the current congestion controller state.

Giving preference to the transmission of its own management frames ensures that the protocol functions efficiently. That is, prioritizing frames other than STREAM frames ensures that loss recovery, congestion control, and flow control operate effectively.

Stream 0 MUST be prioritized over other streams prior to the completion of the cryptographic handshake. This includes the retransmission of the second flight of client handshake messages, that is, the TLS Finished and any client authentication messages.

STREAM data in frames determined to be lost SHOULD be retransmitted before sending new data, unless application priorities indicate otherwise. Retransmitting lost stream data can fill in gaps, which allows the peer to consume already received data and free up flow control window.

10. Flow Control

It is necessary to limit the amount of data that a sender may have outstanding at any time, so as to prevent a fast sender from overwhelming a slow receiver, or to prevent a malicious sender from consuming significant resources at a receiver. This section describes QUIC's flow-control mechanisms.

QUIC employs a credit-based flow-control scheme similar to HTTP/2's flow control [HTTP2]. A receiver advertises the number of octets it is prepared to receive on a given stream and for the entire connection. This leads to two levels of flow control in QUIC: (i)

Connection flow control, which prevents senders from exceeding a receiver's buffer capacity for the connection, and (ii) Stream flow control, which prevents a single stream from consuming the entire receive buffer for a connection.

A data receiver sends MAX_STREAM_DATA or MAX_DATA frames to the sender to advertise additional credit. MAX_STREAM_DATA frames send the the maximum absolute byte offset of a stream, while MAX_DATA sends the maximum sum of the absolute byte offsets of all streams other than stream 0.

A receiver MAY advertise a larger offset at any point by sending MAX_DATA or MAX_STREAM_DATA frames. A receiver MUST NOT renege on an advertisement; that is, once a receiver advertises an offset, it MUST NOT subsequently advertise a smaller offset. A sender could receive MAX_DATA or MAX_STREAM_DATA frames out of order; a sender MUST therefore ignore any flow control offset that does not move the window forward.

A receiver MUST close the connection with a FLOW_CONTROL_ERROR error (Section 11) if the peer violates the advertised connection or stream data limits.

A sender SHOULD send BLOCKED or STREAM_BLOCKED frames to indicate it has data to write but is blocked by flow control limits. These frames are expected to be sent infrequently in common cases, but they are considered useful for debugging and monitoring purposes.

A receiver advertises credit for a stream by sending a MAX_STREAM_DATA frame with the Stream ID set appropriately. A receiver could use the current offset of data consumed to determine the flow control offset to be advertised. A receiver MAY send MAX_STREAM_DATA frames in multiple packets in order to make sure that the sender receives an update before running out of flow control credit, even if one of the packets is lost.

Connection flow control is a limit to the total bytes of stream data sent in STREAM frames on all streams except stream 0. A receiver advertises credit for a connection by sending a MAX_DATA frame. A receiver maintains a cumulative sum of bytes received on all contributing streams, which are used to check for flow control violations. A receiver might use a sum of bytes consumed on all contributing streams to determine the maximum data limit to be advertised.

10.1. Edge Cases and Other Considerations

There are some edge cases which must be considered when dealing with stream and connection level flow control. Given enough time, both endpoints must agree on flow control state. If one end believes it can send more than the other end is willing to receive, the connection will be torn down when too much data arrives.

Conversely if a sender believes it is blocked, while endpoint B expects more data can be received, then the connection can be in a deadlock, with the sender waiting for a MAX_DATA or MAX_STREAM_DATA frame which will never come.

On receipt of a RST_STREAM frame, an endpoint will tear down state for the matching stream and ignore further data arriving on that stream. This could result in the endpoints getting out of sync, since the RST_STREAM frame may have arrived out of order and there may be further bytes in flight. The data sender would have counted the data against its connection level flow control budget, but a receiver that has not received these bytes would not know to include them as well. The receiver must learn the number of bytes that were sent on the stream to make the same adjustment in its connection flow controller.

To avoid this de-synchronization, a RST_STREAM sender MUST include the final byte offset sent on the stream in the RST_STREAM frame. On receiving a RST_STREAM frame, a receiver definitively knows how many bytes were sent on that stream before the RST_STREAM frame, and the receiver MUST use the final offset to account for all bytes sent on the stream in its connection level flow controller.

10.1.1. Response to a RST_STREAM

RST_STREAM terminates one direction of a stream abruptly. Whether any action or response can or should be taken on the data already received is an application-specific issue, but it will often be the case that upon receipt of a RST_STREAM an endpoint will choose to stop sending data in its own direction. If the sender of a RST_STREAM wishes to explicitly state that no future data will be processed, that endpoint MAY send a STOP_SENDING frame at the same time.

10.1.2. Data Limit Increments

This document leaves when and how many bytes to advertise in a MAX_DATA or MAX_STREAM_DATA to implementations, but offers a few considerations. These frames contribute to connection overhead. Therefore frequently sending frames with small changes is

undesirable. At the same time, infrequent updates require larger increments to limits if blocking is to be avoided. Thus, larger updates require a receiver to commit to larger resource commitments. Thus there is a tradeoff between resource commitment and overhead when determining how large a limit is advertised.

A receiver MAY use an autotuning mechanism to tune the frequency and amount that it increases data limits based on a round-trip time estimate and the rate at which the receiving application consumes data, similar to common TCP implementations.

10.1.3. Handshake Exemption

During the initial handshake, an endpoint could need to send a larger message on stream 0 than would ordinarily be permitted by the peer's initial stream flow control window. Since MAX_STREAM_DATA frames are not permitted in these early packets, the peer cannot provide additional flow control window in order to complete the handshake.

Endpoints MAY exceed the flow control limits on stream 0 prior to the completion of the cryptographic handshake. (That is, in Initial, Retry, and Handshake packets.) However, once the handshake is complete, endpoints MUST NOT send additional data beyond the peer's permitted offset. If the amount of data sent during the handshake exceeds the peer's maximum offset, the endpoint cannot send additional data on stream 0 until the peer has sent a MAX_STREAM_DATA frame indicating a larger maximum offset.

10.2. Stream Limit Increment

As with flow control, this document leaves when and how many streams to make available to a peer via MAX_STREAM_ID to implementations, but offers a few considerations. MAX_STREAM_ID frames constitute minimal overhead, while withholding MAX_STREAM_ID frames can prevent the peer from using the available parallelism.

Implementations will likely want to increase the maximum stream ID as peer-initiated streams close. A receiver MAY also advance the maximum stream ID based on current activity, system conditions, and other environmental factors.

10.2.1. Blocking on Flow Control

If a sender does not receive a MAX_DATA or MAX_STREAM_DATA frame when it has run out of flow control credit, the sender will be blocked and SHOULD send a BLOCKED or STREAM_BLOCKED frame. These frames are expected to be useful for debugging at the receiver; they do not require any other action. A receiver SHOULD NOT wait for a BLOCKED

or STREAM_BLOCKED frame before sending MAX_DATA or MAX_STREAM_DATA, since doing so will mean that a sender is unable to send for an entire round trip.

For smooth operation of the congestion controller, it is generally considered best to not let the sender go into quiescence if avoidable. To avoid blocking a sender, and to reasonably account for the possibiity of loss, a receiver should send a MAX_DATA or MAX_STREAM_DATA frame at least two round trips before it expects the sender to get blocked.

A sender sends a single BLOCKED or STREAM_BLOCKED frame only once when it reaches a data limit. A sender SHOULD NOT send multiple BLOCKED or STREAM_BLOCKED frames for the same data limit, unless the original frame is determined to be lost. Another BLOCKED or STREAM_BLOCKED frame can be sent after the data limit is increased.

10.3. Stream Final Offset

The final offset is the count of the number of octets that are transmitted on a stream. For a stream that is reset, the final offset is carried explicitly in a RST_STREAM frame. Otherwise, the final offset is the offset of the end of the data carried in a STREAM frame marked with a FIN flag, or 0 in the case of incoming unidirectional streams.

An endpoint will know the final offset for a stream when the receive stream enters the "Size Known" or "Reset Recvd" state.

An endpoint MUST NOT send data on a stream at or beyond the final offset.

Once a final offset for a stream is known, it cannot change. If a RST_STREAM or STREAM frame causes the final offset to change for a stream, an endpoint SHOULD respond with a FINAL_OFFSET_ERROR error (see Section 11). A receiver SHOULD treat receipt of data at or beyond the final offset as a FINAL_OFFSET_ERROR error, even after a stream is closed. Generating these errors is not mandatory, but only because requiring that an endpoint generate these errors also means that the endpoint needs to maintain the final offset state for closed streams, which could mean a significant state commitment.

11. Error Handling

An endpoint that detects an error SHOULD signal the existence of that error to its peer. Both transport-level and application-level errors can affect an entire connection (see Section 11.1), while only

application-level errors can be isolated to a single stream (see Section 11.2).

The most appropriate error code (Section 11.3) SHOULD be included in the frame that signals the error. Where this specification identifies error conditions, it also identifies the error code that is used.

A stateless reset (Section 6.9.4) is not suitable for any error that can be signaled with a CONNECTION_CLOSE, APPLICATION_CLOSE, or RST_STREAM frame. A stateless reset MUST NOT be used by an endpoint that has the state necessary to send a frame on the connection.

11.1. Connection Errors

Errors that result in the connection being unusable, such as an obvious violation of protocol semantics or corruption of state that affects an entire connection, MUST be signaled using a CONNECTION_CLOSE or APPLICATION_CLOSE frame (Section 7.4, Section 7.5). An endpoint MAY close the connection in this manner even if the error only affects a single stream.

Application protocols can signal application-specific protocol errors using the APPLICATION_CLOSE frame. Errors that are specific to the transport, including all those described in this document, are carried in a CONNECTION_CLOSE frame. Other than the type of error code they carry, these frames are identical in format and semantics.

A CONNECTION_CLOSE or APPLICATION_CLOSE frame could be sent in a packet that is lost. An endpoint SHOULD be prepared to retransmit a packet containing either frame type if it receives more packets on a terminated connection. Limiting the number of retransmissions and the time over which this final packet is sent limits the effort expended on terminated connections.

An endpoint that chooses not to retransmit packets containing CONNECTION_CLOSE or APPLICATION_CLOSE risks a peer missing the first such packet. The only mechanism available to an endpoint that continues to receive data for a terminated connection is to use the stateless reset process (Section 6.9.4).

An endpoint that receives an invalid CONNECTION_CLOSE or APPLICATION_CLOSE frame MUST NOT signal the existence of the error to its peer.

11.2. Stream Errors

If an application-level error affects a single stream, but otherwise leaves the connection in a recoverable state, the endpoint can send a RST_STREAM frame (Section 7.3) with an appropriate error code to terminate just the affected stream.

Stream 0 is critical to the functioning of the entire connection. If stream 0 is closed with either a RST_STREAM or STREAM frame bearing the FIN flag, an endpoint MUST generate a connection error of type PROTOCOL_VIOLATION.

Other than STOPPING (Section 9.3), RST_STREAM MUST be instigated by the application and MUST carry an application error code. Resetting a stream without knowledge of the application protocol could cause the protocol to enter an unrecoverable state. Application protocols might require certain streams to be reliably delivered in order to guarantee consistent state between endpoints.

11.3. Transport Error Codes

QUIC error codes are 16-bit unsigned integers.

This section lists the defined QUIC transport error codes that may be used in a CONNECTION_CLOSE frame. These errors apply to the entire connection.

NO_ERROR (0x0): An endpoint uses this with CONNECTION_CLOSE to signal that the connection is being closed abruptly in the absence of any error.

INTERNAL_ERROR (0x1): The endpoint encountered an internal error and cannot continue with the connection.

SERVER_BUSY (0x2): The server is currently busy and does not accept any new connections.

FLOW_CONTROL_ERROR (0x3): An endpoint received more data than it permitted in its advertised data limits (see Section 10).

STREAM_ID_ERROR (0x4): An endpoint received a frame for a stream identifier that exceeded its advertised maximum stream ID.

STREAM_STATE_ERROR (0x5): An endpoint received a frame for a stream that was not in a state that permitted that frame (see Section 9.2).

FINAL_OFFSET_ERROR (0x6): An endpoint received a STREAM frame containing data that exceeded the previously established final offset. Or an endpoint received a RST_STREAM frame containing a final offset that was lower than the maximum offset of data that was already received. Or an endpoint received a RST_STREAM frame containing a different final offset to the one already established.

FRAME_FORMAT_ERROR (0x7): An endpoint received a frame that was badly formatted. For instance, an empty STREAM frame that omitted the FIN flag, or an ACK frame that has more acknowledgment ranges than the remainder of the packet could carry. This is a generic error code; an endpoint SHOULD use the more specific frame format error codes (0x1XX) if possible.

TRANSPORT_PARAMETER_ERROR (0x8): An endpoint received transport parameters that were badly formatted, included an invalid value, was absent even though it is mandatory, was present though it is forbidden, or is otherwise in error.

VERSION_NEGOTIATION_ERROR (0x9): An endpoint received transport parameters that contained version negotiation parameters that disagreed with the version negotiation that it performed. This error code indicates a potential version downgrade attack.

PROTOCOL_VIOLATION (0xA): An endpoint detected an error with protocol compliance that was not covered by more specific error codes.

UNSOLICITED_PATH_RESPONSE (0xB): An endpoint received a PATH_RESPONSE frame that did not correspond to any PATH_CHALLENGE frame that it previously sent.

FRAME_ERROR (0x1XX): An endpoint detected an error in a specific frame type. The frame type is included as the last octet of the error code. For example, an error in a MAX_STREAM_ID frame would be indicated with the code (0x106).

Codes for errors occuring when TLS is used for the crypto handshake are defined in Section 11 of [QUIC-TLS]. See Section 13.2 for details of registering new error codes.

11.4. Application Protocol Error Codes

Application protocol error codes are 16-bit unsigned integers, but the management of application error codes are left to application protocols. Application protocol error codes are used for the RST_STREAM (Section 7.3) and APPLICATION_CLOSE (Section 7.5) frames.

There is no restriction on the use of the 16-bit error code space for application protocols. However, QUIC reserves the error code with a value of 0 to mean STOPPING. The application error code of STOPPING (0) is used by the transport to cancel a stream in response to receipt of a STOP_SENDING frame.

12. Security and Privacy Considerations

12.1. Spoofed ACK Attack

An attacker might be able to receive an address validation token (Section 6.6) from the server and then release the IP address it used to acquire that token. The attacker may, in the future, spoof this same address (which now presumably addresses a different endpoint), and initiate a 0-RTT connection with a server on the victim's behalf. The attacker can then spoof ACK frames to the server which cause the server to send excessive amounts of data toward the new owner of the IP address.

There are two possible mitigations to this attack. The simplest one is that a server can unilaterally create a gap in packet-number space. In the non-attack scenario, the client will send an ACK frame with the larger value for largest acknowledged. In the attack scenario, the attacker could acknowledge a packet in the gap. If the server sees an acknowledgment for a packet that was never sent, the connection can be aborted.

The second mitigation is that the server can require that acknowledgments for sent packets match the encryption level of the sent packet. This mitigation is useful if the connection has an ephemeral forward-secure key that is generated and used for every new connection. If a packet sent is protected with a forward-secure key, then any acknowledgments that are received for them MUST also be forward-secure protected. Since the attacker will not have the forward secure key, the attacker will not be able to generate forward-secure protected packets with ACK frames.

12.2. Optimistic ACK Attack

An endpoint that acknowledges packets it has not received might cause a congestion controller to permit sending at rates beyond what the network supports. An endpoint MAY skip packet numbers when sending packets to detect this behavior. An endpoint can then immediately close the connection with a connection error of type PROTOCOL_VIOLATION (see Section 6.9.3).

12.3. Slowloris Attacks

The attacks commonly known as Slowloris [SLOWLORIS] try to keep many connections to the target endpoint open and hold them open as long as possible. These attacks can be executed against a QUIC endpoint by generating the minimum amount of activity necessary to avoid being closed for inactivity. This might involve sending small amounts of data, gradually opening flow control windows in order to control the sender rate, or manufacturing ACK frames that simulate a high loss rate.

QUIC deployments SHOULD provide mitigations for the Slowloris attacks, such as increasing the maximum number of clients the server will allow, limiting the number of connections a single IP address is allowed to make, imposing restrictions on the minimum transfer speed a connection is allowed to have, and restricting the length of time an endpoint is allowed to stay connected.

12.4. Stream Fragmentation and Reassembly Attacks

An adversarial endpoint might intentionally fragment the data on stream buffers in order to cause disproportionate memory commitment. An adversarial endpoint could open a stream and send some STREAM frames containing arbitrary fragments of the stream content.

The attack is mitigated if flow control windows correspond to available memory. However, some receivers will over-commit memory and advertise flow control offsets in the aggregate that exceed actual available memory. The over-commitment strategy can lead to better performance when endpoints are well behaved, but renders endpoints vulnerable to the stream fragmentation attack.

QUIC deployments SHOULD provide mitigations against the stream fragmentation attack. Mitigations could consist of avoiding over-committing memory, delaying reassembly of STREAM frames, implementing heuristics based on the age and duration of reassembly holes, or some combination.

12.5. Stream Commitment Attack

An adversarial endpoint can open lots of streams, exhausting state on an endpoint. The adversarial endpoint could repeat the process on a large number of connections, in a manner similar to SYN flooding attacks in TCP.

Normally, clients will open streams sequentially, as explained in Section 9.1. However, when several streams are initiated at short intervals, transmission error may cause STREAM DATA frames opening

streams to be received out of sequence. A receiver is obligated to open intervening streams if a higher-numbered stream ID is received. Thus, on a new connection, opening stream 2000001 opens 1 million streams, as required by the specification.

The number of active streams is limited by the concurrent stream limit transport parameter, as explained in Section 9.4. If chosen judisciously, this limit mitigates the effect of the stream commitment attack. However, setting the limit too low could affect performance when applications expect to open large number of streams.

13. IANA Considerations

13.1. QUIC Transport Parameter Registry

IANA [SHALL add/has added] a registry for "QUIC Transport Parameters" under a "QUIC Protocol" heading.

The "QUIC Transport Parameters" registry governs a 16-bit space. This space is split into two spaces that are governed by different policies. Values with the first byte in the range 0x00 to 0xfe (in hexadecimal) are assigned via the Specification Required policy [RFC8126]. Values with the first byte 0xff are reserved for Private Use [RFC8126].

Registrations MUST include the following fields:

Value: The numeric value of the assignment (registrations will be between 0x0000 and 0xfeff).

Parameter Name: A short mnemonic for the parameter.

Specification: A reference to a publicly available specification for the value.

The nominated expert(s) verify that a specification exists and is readily accessible. The expert(s) are encouraged to be biased towards approving registrations unless they are abusive, frivolous, or actively harmful (not merely aesthetically displeasing, or architecturally dubious).

The initial contents of this registry are shown in Table 7.

Value	Parameter Name	Specification
0x0000	initial_max_stream_data	Section 6.4.1
0x0001	initial_max_data	Section 6.4.1
0x0002	initial_max_stream_id_bidi	Section 6.4.1
0x0003	idle_timeout	Section 6.4.1
0x0005	max_packet_size	Section 6.4.1
0x0006	stateless_reset_token	Section 6.4.1
0x0007	ack_delay_exponent	Section 6.4.1
0x0008	initial_max_stream_id_uni	Section 6.4.1

Table 7: Initial QUIC Transport Parameters Entries

13.2. QUIC Transport Error Codes Registry

IANA [SHALL add/has added] a registry for "QUIC Transport Error Codes" under a "QUIC Protocol" heading.

The "QUIC Transport Error Codes" registry governs a 16-bit space. This space is split into two spaces that are governed by different policies. Values with the first byte in the range 0x00 to 0xfe (in hexadecimal) are assigned via the Specification Required policy [RFC8126]. Values with the first byte 0xff are reserved for Private Use [RFC8126].

Registrations MUST include the following fields:

Value: The numeric value of the assignment (registrations will be between 0x0000 and 0xfeff).

Code: A short mnemonic for the parameter.

Description: A brief description of the error code semantics, which MAY be a summary if a specification reference is provided.

Specification: A reference to a publicly available specification for the value.

The initial contents of this registry are shown in Table 8. Note that FRAME_ERROR takes the range from 0x100 to 0x1FF and private use occupies the range from 0xFE00 to 0xFFFF.

Value	Error	Description	Specification
0x0	NO_ERROR	No error	Section 11.3
0x1	INTERNAL_ERROR	Implementation error	Section 11.3
0x2	SERVER_BUSY	Server currently busy	Section 11.3
0x3	FLOW_CONTROL_ERROR	Flow control error	Section 11.3
0x4	STREAM_ID_ERROR	Invalid stream ID	Section 11.3
0x5	STREAM_STATE_ERROR	Frame received in invalid stream state	Section 11.3
0x6	FINAL_OFFSET_ERROR	Change to final stream offset	Section 11.3
0x7	FRAME_FORMAT_ERROR	Generic frame format error	Section 11.3
0x8	TRANSPORT_PARAMETER_ERROR	Error in transport parameters	Section 11.3
0x9	VERSION_NEGOTIATION_ERROR	Version negotiation failure	Section 11.3
0xA	PROTOCOL_VIOLATION	Generic protocol violation	Section 11.3
0xB	UNSOLICITED_PATH_RESPO	Unsolicited	Section 11.3

	NSE	PATH_RESPONSE frame		
0x100-0x1FF	FRAME_ERROR	Specific frame format error	Section 11.3	

Table 8: Initial QUIC Transport Error Codes Entries

14. References

14.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-21 (work in progress), July 2017.

[PLPMTUD] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <https://www.rfc-editor.org/info/rfc4821>.

[PMTUDv4] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <https://www.rfc-editor.org/info/rfc1191>.

[PMTUDv6] McCann, J., Deering, S., Mogul, J., and R. Hinden, Ed., "Path MTU Discovery for IP version 6", STD 87, RFC 8201, DOI 10.17487/RFC8201, July 2017, <https://www.rfc-editor.org/info/rfc8201>.

[QUIC-RECOVERY]
Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", draft-ietf-quic-recovery-10 (work in progress), April 2018.

[QUIC-TLS]
Thomson, M., Ed. and S. Turner, Ed., "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-10 (work in progress), April 2018.

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <https://www.rfc-editor.org/info/rfc1191>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <https://www.rfc-editor.org/info/rfc4821>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

14.2. Informative References

[EARLY-DESIGN] Roskind, J., "QUIC: Multiplexed Transport Over UDP", December 2013, <https://goo.gl/dMVtFi>.

[HTTP2] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[QUIC-INVARIANTS] Thomson, M., "Version-Independent Properties of QUIC", draft-ietf-quic-invariants-01 (work in progress), April 2018.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[RFC2360] Scott, G., "Guide for Internet Standards Writers", BCP 22, RFC 2360, DOI 10.17487/RFC2360, June 1998, <https://www.rfc-editor.org/info/rfc2360>.

[RFC4787] Audet, F., Ed. and C. Jennings, "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP", BCP 127, RFC 4787, DOI 10.17487/RFC4787, January 2007, <https://www.rfc-editor.org/info/rfc4787>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[SLOWLORIS] RSnake Hansen, R., "Welcome to Slowloris...", June 2009, <https://web.archive.org/web/20150315054838/http://ha.ckers.org/slowloris/>.

[SST] Ford, B., "Structured streams", ACM SIGCOMM Computer Communication Review Vol. 37, pp. 361, DOI 10.1145/1282427.1282421, October 2007.

14.3. URIs

[1] https://mailarchive.ietf.org/arch/search/?email_list=quic

[2] https://github.com/quicwg

[3] https://github.com/quicwg/base-drafts/labels/-transport

[4] https://github.com/quicwg/base-drafts/wiki/QUIC-Versions

Appendix A. Contributors

The original authors of this specification were Ryan Hamilton, Jana Iyengar, Ian Swett, and Alyssa Wilk.

The original design and rationale behind this protocol draw significantly from work by Jim Roskind [EARLY-DESIGN]. In alphabetical order, the contributors to the pre-IETF QUIC project at Google are: Britt Cyr, Jeremy Dorfman, Ryan Hamilton, Jana Iyengar, Fedor Kouranov, Charles Krasic, Jo Kulik, Adam Langley, Jim Roskind,

Robbie Shade, Satyam Shekhar, Cherie Shi, Ian Swett, Raman Tenneti, Victor Vasiliev, Antonio Vicente, Patrik Westin, Alyssa Wilk, Dale Worley, Fan Yang, Dan Zhang, Daniel Ziegler.

Appendix B. Acknowledgments

Special thanks are due to the following for helping shape pre-IETF QUIC and its deployment: Chris Bentzel, Misha Efimov, Roberto Peon, Alistair Riddoch, Siddharth Vijayakrishnan, and Assar Westerlund.

This document has benefited immensely from various private discussions and public ones on the quic@ietf.org and proto-quic@chromium.org mailing lists. Our thanks to all.

Appendix C. Change Log

> *RFC Editor's Note:* Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

C.1. Since draft-ietf-quic-transport-10

- Swap payload length and packed number fields in long header (#1294)
- Clarified that CONNECTION_CLOSE is allowed in Handshake packet (#1274)
- Spin bit reserved (#1283)
- Coalescing multiple QUIC packets in a UDP datagram (#1262, #1285)
- A more complete connection migration (#1249)
- Refine opportunistic ACK defense text (#305, #1030, #1185)
- A Stateless Reset Token isn't mandatory (#818, #1191)
- Removed implicit stream opening (#896, #1193)
- An empty STREAM frame can be used to open a stream without sending data (#901, #1194)
- Define stream counts in transport parameters rather than a maximum stream ID (#1023, #1065)
- STOP_SENDING is now prohibited before streams are used (#1050)

- Recommend including ACK in Retry packets and allow PADDING (#1067, #882)
- Endpoints now become closing after an idle timeout (#1178, #1179)
- Remove implication that Version Negotiation is sent when a packet of the wrong version is received (#1197)

C.2. Since draft-ietf-quic-transport-09

- Added PATH_CHALLENGE and PATH_RESPONSE frames to replace PING with Data and PONG frame. Changed ACK frame type from 0x0e to 0x0d. (#1091, #725, #1086)
- A server can now only send 3 packets without validating the client address (#38, #1090)
- Delivery order of stream data is no longer strongly specified (#252, #1070)
- Rework of packet handling and version negotiation (#1038)
- Stream 0 is now exempt from flow control until the handshake completes (#1074, #725, #825, #1082)
- Improved retransmission rules for all frame types: information is retransmitted, not packets or frames (#463, #765, #1095, #1053)
- Added an error code for server busy signals (#1137)
- Endpoints now set the connection ID that their peer uses. Connection IDs are variable length. Removed the omit_connection_id transport parameter and the corresponding short header flag. (#1089, #1052, #1146, #821, #745, #821, #1166, #1151)

C.3. Since draft-ietf-quic-transport-08

- Clarified requirements for BLOCKED usage (#65, #924)
- BLOCKED frame now includes reason for blocking (#452, #924, #927, #928)
- GAP limitation in ACK Frame (#613)
- Improved PMTUD description (#614, #1036)
- Clarified stream state machine (#634, #662, #743, #894)

- Reserved versions don’t need to be generated deterministically (#831, #931)
- You don’t always need the draining period (#871)
- Stateless reset clarified as version-specific (#930, #986)
- initial_max_stream_id_x transport parameters are optional (#970, #971)
- Ack Delay assumes a default value during the handshake (#1007, #1009)
- Removed transport parameters from NewSessionTicket (#1015)

C.4. Since draft-ietf-quic-transport-07

- The long header now has version before packet number (#926, #939)
- Rename and consolidate packet types (#846, #822, #847)
- Packet types are assigned new codepoints and the Connection ID Flag is inverted (#426, #956)
- Removed type for Version Negotiation and use Version 0 (#963, #968)
- Streams are split into unidirectional and bidirectional (#643, #656, #720, #872, #175, #885)
  - Stream limits now have separate uni- and bi-directinal transport parameters (#909, #958)
  - Stream limit transport parameters are now optional and default to 0 (#970, #971)
- The stream state machine has been split into read and write (#634, #894)
- Employ variable-length integer encodings throughout (#595)
- Improvements to connection close
  - Added distinct closing and draining states (#899, #871)
  - Draining period can terminate early (#869, #870)
  - Clarifications about stateless reset (#889, #890)

- Address validation for connection migration (#161, #732, #878)
- Clearly defined retransmission rules for BLOCKED (#452, #65, #924)
- negotiated_version is sent in server transport parameters (#710, #959)
- Increased the range over which packet numbers are randomized (#864, #850, #964)

C.5. Since draft-ietf-quic-transport-06

- Replaced FNV-1a with AES-GCM for all "Cleartext" packets (#554)
- Split error code space between application and transport (#485)
- Stateless reset token moved to end (#820)
- 1-RTT-protected long header types removed (#848)
- No acknowledgments during draining period (#852)
- Remove "application close" as a separate close type (#854)
- Remove timestamps from the ACK frame (#841)
- Require transport parameters to only appear once (#792)

C.6. Since draft-ietf-quic-transport-05

- Stateless token is server-only (#726)
- Refactor section on connection termination (#733, #748, #328, #177)
- Limit size of Version Negotiation packet (#585)
- Clarify when and what to ack (#736)
- Renamed STREAM_ID_NEEDED to STREAM_ID_BLOCKED
- Clarify Keep-alive requirements (#729)

C.7. Since draft-ietf-quic-transport-04

- Introduce STOP_SENDING frame, RST_STREAM only resets in one direction (#165)

- Removed GOAWAY; application protocols are responsible for graceful shutdown (#696)
- Reduced the number of error codes (#96, #177, #184, #211)
- Version validation fields can’t move or change (#121)
- Removed versions from the transport parameters in a NewSessionTicket message (#547)
- Clarify the meaning of "bytes in flight" (#550)
- Public reset is now stateless reset and not visible to the path (#215)
- Reordered bits and fields in STREAM frame (#620)
- Clarifications to the stream state machine (#572, #571)
- Increased the maximum length of the Largest Acknowledged field in ACK frames to 64 bits (#629)
- truncate_connection_id is renamed to omit_connection_id (#659)
- CONNECTION_CLOSE terminates the connection like TCP RST (#330, #328)
- Update labels used in HKDF-Expand-Label to match TLS 1.3 (#642)

C.8. Since draft-ietf-quic-transport-03

- Change STREAM and RST_STREAM layout
- Add MAX_STREAM_ID settings

C.9. Since draft-ietf-quic-transport-02

- The size of the initial packet payload has a fixed minimum (#267, #472)
- Define when Version Negotiation packets are ignored (#284, #294, #241, #143, #474)
- The 64-bit FNV-1a algorithm is used for integrity protection of unprotected packets (#167, #480, #481, #517)
- Rework initial packet types to change how the connection ID is chosen (#482, #442, #493)

- No timestamps are forbidden in unprotected packets (#542, #429)
- Cryptographic handshake is now on stream 0 (#456)
- Remove congestion control exemption for cryptographic handshake (#248, #476)
- Version 1 of QUIC uses TLS; a new version is needed to use a different handshake protocol (#516)
- STREAM frames have a reduced number of offset lengths (#543, #430)
- Split some frames into separate connection- and stream- level frames (#443)
  - WINDOW_UPDATE split into MAX_DATA and MAX_STREAM_DATA (#450)
  - BLOCKED split to match WINDOW_UPDATE split (#454)
  - Define STREAM_ID_NEEDED frame (#455)
- A NEW_CONNECTION_ID frame supports connection migration without linkability (#232, #491, #496)
- Transport parameters for 0-RTT are retained from a previous connection (#405, #513, #512)
  - A client in 0-RTT no longer required to reset excess streams (#425, #479)
- Expanded security considerations (#440, #444, #445, #448)

C.10. Since draft-ietf-quic-transport-01

- Defined short and long packet headers (#40, #148, #361)
- Defined a versioning scheme and stable fields (#51, #361)
- Define reserved version values for "greasing" negotiation (#112, #278)
- The initial packet number is randomized (#35, #283)
- Narrow the packet number encoding range requirement (#67, #286, #299, #323, #356)
- Defined client address validation (#52, #118, #120, #275)

- Define transport parameters as a TLS extension (#49, #122)
- SCUP and COPT parameters are no longer valid (#116, #117)
- Transport parameters for 0-RTT are either remembered from before, or assume default values (#126)
- The server chooses connection IDs in its final flight (#119, #349, #361)
- The server echoes the Connection ID and packet number fields when sending a Version Negotiation packet (#133, #295, #244)
- Defined a minimum packet size for the initial handshake packet from the client (#69, #136, #139, #164)
- Path MTU Discovery (#64, #106)
- The initial handshake packet from the client needs to fit in a single packet (#338)
- Forbid acknowledgment of packets containing only ACK and PADDING (#291)
- Require that frames are processed when packets are acknowledged (#381, #341)
- Removed the STOP_WAITING frame (#66)
- Don’t require retransmission of old timestamps for lost ACK frames (#308)
- Clarified that frames are not retransmitted, but the information in them can be (#157, #298)
- Error handling definitions (#335)
- Split error codes into four sections (#74)
- Forbid the use of Public Reset where CONNECTION_CLOSE is possible (#289)
- Define packet protection rules (#336)
- Require that stream be entirely delivered or reset, including acknowledgment of all STREAM frames or the RST_STREAM, before it closes (#381)

- Remove stream reservation from state machine (#174, #280)
- Only stream 1 does not contribute to connection-level flow control (#204)
- Stream 1 counts towards the maximum concurrent stream limit (#201, #282)
- Remove connection-level flow control exclusion for some streams (except 1) (#246)
- RST_STREAM affects connection-level flow control (#162, #163)
- Flow control accounting uses the maximum data offset on each stream, rather than bytes received (#378)
- Moved length-determining fields to the start of STREAM and ACK (#168, #277)
- Added the ability to pad between frames (#158, #276)
- Remove error code and reason phrase from GOAWAY (#352, #355)
- GOAWAY includes a final stream number for both directions (#347)
- Error codes for RST_STREAM and CONNECTION_CLOSE are now at a consistent offset (#249)
- Defined priority as the responsibility of the application protocol (#104, #303)

C.11. Since draft-ietf-quic-transport-00

- Replaced DIVERSIFICATION_NONCE flag with KEY_PHASE flag
- Defined versioning
- Reworked description of packet and frame layout
- Error code space is divided into regions for each component
- Use big endian for all numeric values

C.12. Since draft-hamilton-quic-transport-protocol-01

- Adopted as base for draft-ietf-quic-tls
- Updated authors/editors list

- Added IANA Considerations section
- Moved Contributors and Acknowledgments to appendices

Authors' Addresses

Jana Iyengar (editor)
Fastly

Email: jri.ietf@gmail.com

Martin Thomson (editor)
Mozilla

Email: martin.thomson@gmail.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 25, 2017

I. Johansson
Ericsson AB

February 21, 2017

ECN support in QUIC
draft-johansson-quic-ecn-01

Abstract

This memo outlines the ECN support in QUIC. The intention is that most of the material ends up updating other new or existing QUIC protocol specifications, thus it may be possible that this draft does not warrant a working group status.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 25, 2017.

Copyright Notice

Table of Contents

1. Introduction

ECN support in transport protocols is a fundamental feature that should be included in the QUIC specification as a mandatory element. The benefits of ECN is described in [I-D.ietf-aqm-ecn-benefits]. The ECN support should be implemented to support both present and future ECN, the latter is outlined in [I-D.ietf-tsvwg-ecn-experimentation], of particular interest is the ability to discriminate between classic ECN and L4S ECN by means of differentiation between the use of the ECT(0) and ECT(1) code points. This draft does however not delve into the details of the congestion control implementation.

2. Elements of ECN support

This draft covers the following aspects of ECN support:

- ECN negotiation
- ECN echo
- ECN bits in the IP header, semantics
- Fallback in case of ECN fault

- OS socket specifics, access to the ECN bits
- Monitoring

2.1. ECN negotialtion

ECN support in QUIC needs to be negotiated. The reasons is that network elements may not support ECN and may either clear the ECN bits or simply discard packets that have the ECN bits set. In addition, a QUIC implementation may not have access to the ECN bits in the IP header due to OS dependent restrictions, investigations (Piers O'Hanlon) have indicated that this is in certain cases an asymmetric property, for instance while it is possible to set the ECN bits it is not possible to read them.

It is also required that the ECN negotiation does not interfere with the connection setup, in other words a failed ECN negotation should not cause an extra roundtrip for the connection setup.

The suggested method in this draft is to add an ECN negotiation frame that is transmitted when connection setup is completed. Both peers MUST transmit the ECN negotation frame. The ECN negotiation frame is shown below.

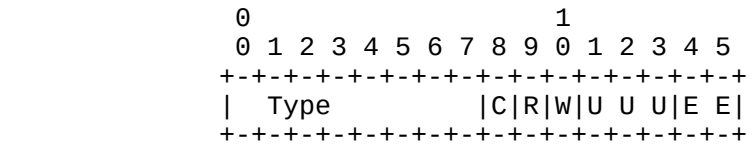


Figure 1: ECN negotation frame

The 2nd byte contains the flags:

- C: Challenge bit, indicates that the transmitted ECN negotiation frame is a challenge, if bit is not set then it is a response.
- R: Possible to read ECN bits in IP header
- W: Possible to write ECN bits in IP header
- EE : Echo of ECN bits
- U: Unused

A peer transmits the ECN negotiation frame with the R,W and EE bits in the 2nd byte set to '0' and the C bit set to '1'. This frame is echoed back with the flags set occording to the degree of ECN support

and with the ECN bits in the IP header of the received ECN negotiation frame copied to the EE field, the C bit is '0'. As both peers MUST transmit an ECN negotation frame there will be a total of 4 ECN negotiation frames transmitted, two challenges and two responses.

The IP header for the ECN negotiation frame should set the ECN bits to CE '11'. When the corresponding response is received then an EE pattern of '11' indicates that ECN is likely supported in the network. This does not give a full quarantee that ECN is supported in the network. Monitoring of the ECN field in the ACK-frame serves to give further indication of ECN support once ECN is turned on.

A peer is not allowed to set ECT on outgoing data packets until a ECN negotiation response that inticates that ECN is supported is received. In other words it is only the ECN negotiation frame that is allowed to set the ECN bits in the IP header.

A lack of an ECN negotiation response may indicate that the ECN challenge frame or the ECN response frame was lost or that a node in the network deliberately discards ECN-CE marked packets. The peer can transmit additonal ECN challeges with given time intervals to rule out accidentail packet loss. The detailed timing for this is T.B.D.

The mode mechanism in [RFC6679] can serve as in input to a solution for the support of ECN in the case that OS ECN support is asymmetric. It is however unclear how a QUIC implementation can determine asymmetric ECN support in the underlying OS. For instance the method to send ECN marked packets to the local host to determine OS support does not reveal if the OS ECN support is asymmetric.

2.2. ECN bits in the IP header, semantics

The ECN bits in the IP header should be set according to the recommendations in [I-D.ietf-tsvwg-ecn-experimentation]. This means that the meaning of ECT(0) and ECT(1) differ.

2.3. ECN echo

The ECN echo should prefferably go into the ACK frame [I-D.ietf-quic-transport], this is beneficial as the ECN information can then use some of the already existing data in the ACK frame for improved efficiency, this applies especially to alternatives 1 and 2 below. It is suggested that the 'U' bit in the ACK frame type is renamed 'E' to indicate the presence of an ECN field in the ACK frame, this makes it possible to omit the ECN information for the cases where ECN is not supported for the connection.

Currently there are three alternatives how to add ECN support to the ACK frames .

The first alternative inserts a one octet field that contains a 2 bit ECN echo, followed by the ACK block length. The ACK block length then dictates the number of received contiguous frames with the indicated ECN echo.

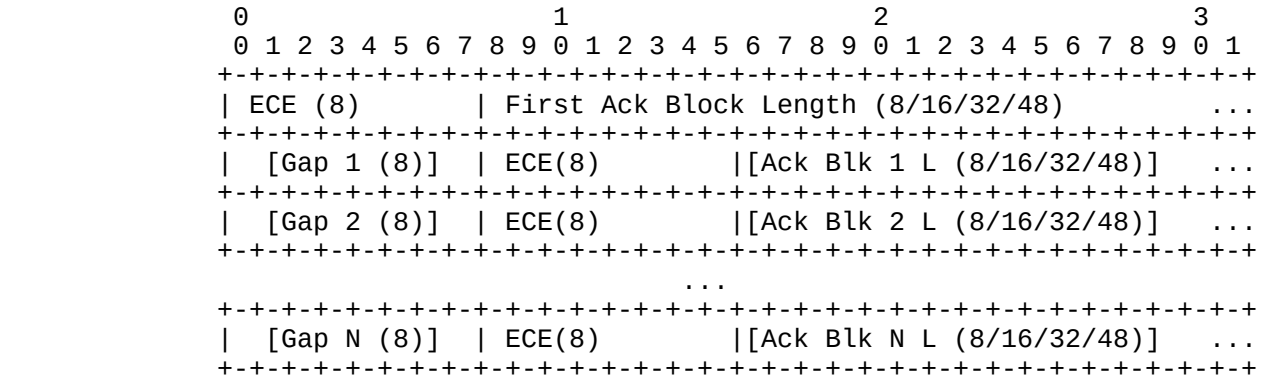


Figure 2: ECN field in ACK frame ACK block, alt 1

The second alternative encodes a variable length field that contains the ECN echoes for the frames listed in the ACK blocks. The length of the field is inferred from the ACK block lengths. No ECN echoes are indicated for the gaps (it is, after all, impossible to indicate status of the ECN bits for lost packets). For instance if the ACK blocks list 10 frames, then the length of the ECN echo field becomes 2*10=20bits, with additional 4 bits of padding the ECN echo field will then become 3 octets long.

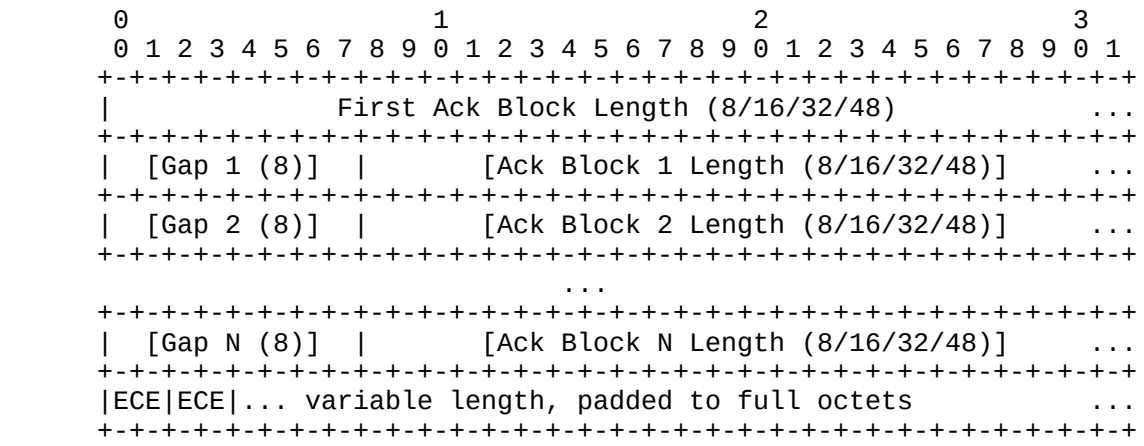


Figure 3: ECN field in ACK frame ACK block, alt 2

The third alternative encodes the number of bytes that are marked ECT(0), ECT(1) and CE with 32 bits each, the total extra overhead is thus 12 octets.

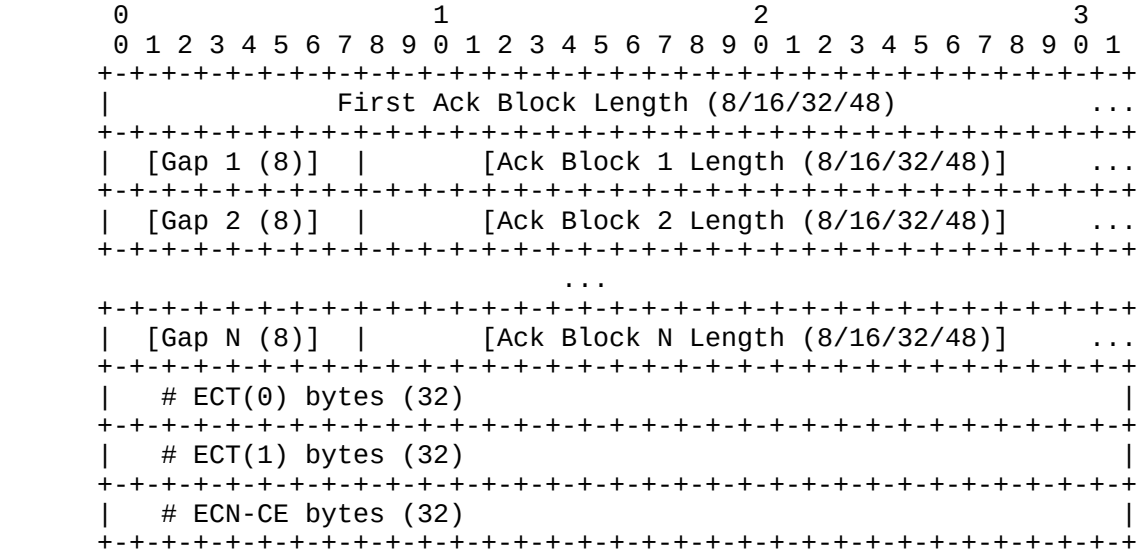


Figure 4: ECN field in ACK frame ACK block, alt 3

The fourth alternative use an extra byte to encode how many bits that encode each of the ECT/CE fields.

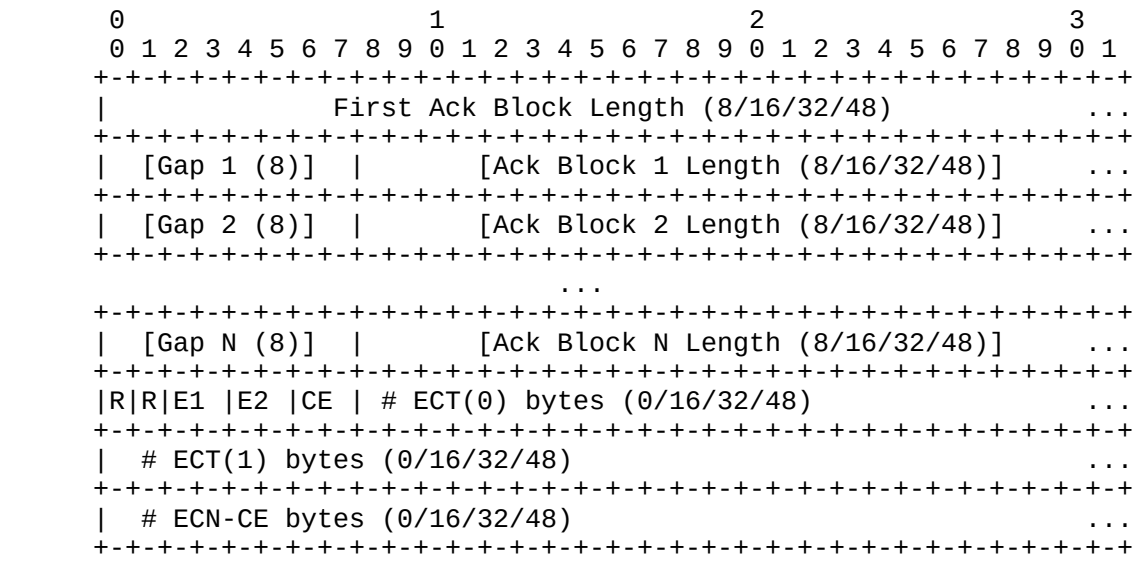


Figure 5: ECN field in ACK frame ACK block, alt 4

The E1,E2 and CE fields indicate the length of each encoding for the number of ECT(0), ECT(1) and ECN-CE marked bytes. This is encoded as:

- 00: 0 bits
- 01: 16 bits
- 10: 32 bits
- 11: 48bits

R indicates reserved bits.

There are pros an cons with the four alternatives:

- Alt 1: Is very compact in the case where the ECN bits are largely unchanged. However in the worst case where received frames flip forth and back between ECT and CE then each frame will require at least 3 octets overhead (ECE, ACK block length, Gap).
- Alt 2: Is quite compact as it only requires two bits encoding per frame. The additional overhead amounts to ceil(N*2/8) octets where the N is the sum of the ACK block lengths. On the downside is that it is a less efficient format for the case that the ECN

bits are unchanged. One uncertainty is if STOP_WAITING frames could make this encoding bulky.

- Alt 3: Has a fixed 12 octet overhead which may be beneficial as it gives a deterministic overhead. The possible drawback is that it is not possible to know exactly which frames have been remarked, something that can limit the ability to detect network ECN faults based on the method to transmit a pattern on ECT and CE marked packets.

- Alt 4: Is a variation to Alt 3 but has a variable length encoding that should consume less space, especially in the cases that one of the ECT code points is not used and for the case that packets are only sporadically ECN-CE marked. This alternative also makes it unnecessary to use a bit in the ACK frame type to indicate the precense of an ECN field as this can be indicated in a efficient way with the one byte header in this format. E0=E1=CE = 00 indicates that the following ECT and CE fields are encoded with zero bits.

Which of the three formats above (or something else) that is the best alternative is subject to discussion.

2.4. Fallback in case of ECN fault

ECN can be subject to issues in network equipment, such as remarking to Not-ECN, remarking from ECT(0) to ECT(1) and vice versa or constant remarking to ECN-CE. Furthermore ECT marked packets may be discarded in the network. While these problems seem to be rare, see for instance [McQuistin-Perkins], it is still necessary to safeguard against such problems.

A peer should disable ECN for its outgoing packets if ECN fault is detected, it is however still possible for the other peer to use ECN.

TODO add more information as regards to how to detect network ECN faults. [ECN-fallback](expired) gives a few examples for fault detection. Examples on how to detect ECN faults include for instance the method to set ECT and CE for outgoing packets according to a given pattern.

Fallback in case of ECN faults is not an issue only for QUIC, it is here suggested that mechanisms for this is described in a non QUIC related draft, for instance in TSVWG.

2.5. OS socket specifics, access to the ECN bits

ECN support in QUIC comes with the additional challenge that it is necessary to somehow access the ECN bits in the IP headers. In TCP this is provided without major concerns as TCP is generally implemented in OS kernel space. QUIC can however be implemented both in user space or kernel space and is layered on top of UDP, which means that access to the ECN bits is not a given, instead various tricks are needed.

The text below is copy-pasted from [OHanlon].

"To set ECN on Linux, BSD and OSX one can use IP_TOS socket option, with the setsockopt() call, to set the relevant ECN bits of the TOS byte. On Windows one can use a similar technique though firstly one has to enable TOS byte setting by enabling a particular Registry key (DisableUserTOSSetting=0 (see https://msdn.microsoft.com/en-us/library/windows/desktop/dd874008%28v=vs.85%29.aspx One could also probably use the libpcap write functionality."

"To obtain the ECN bits from a packet one needs a mechanism to retrieve the ECN bits from each packet. On Linux, one needs to firstly set the IP_RECVTOS socket option on the receiving socket, and use the recvmsg() call to receive a packet, and then retrieve the TOS byte from the associated csmg structure returned by the recvmsg() call. This still works with linux-4.2.3. On OSX/BSD there are no suitable socket options to retrieve the ECN/TOS bits and one cannot use raw sockets as they do not function for UDP/TCP sockets (they do work with ICMP), so one has to use alternatives such the bpf interface, or a REDIRECT socket. Whilst on Windows it seems that the only way to retrieve the ECN bits is via a raw socket, or custom NDIS driver, though it’s possible there’s an API I’m missing."

TODO: Write a more detailed description on how to implement ECN support in QUIC for different OS stacks.

2.6. Monitoring

A QUIC implementation should monitor the ECN functionality in order to provide input to e.g. service providers to improve ECN support in the networks. Items of interest are:

- Black holes, ECT or CE marked packets are discarded.
- Faulty remarking, e.g. ECT(0) is remarked to ECT(1) or Not-ECT.
- Continuous CE marking, possible indication of faulty on/off ECN marking, but can also be an effect of severe congestion.

- Degree of L4S support. L4S should generally give low queue latency. Estimation of one way queue delay for L4S enabled QUIC connections can be used to determine if there are congested nodes along the path that are not L4S capable.

3. IANA Considerations

T.B.D.

4. Open questions

A list of open questions:

- Is it sufficient that one peer sends an ECN negotiation challenge frame?.
- Should the ECN field in the ACK frame be mandatory ? (in which case it is not necessary to indicate its presence)
- Should all packets be ECT or should there be special patterns to improve fault detection.
- Write up a more detailed description on how to implement ECN support in QUIC for different OS stacks.
- Determine which ECN echo encoding in the ACK frame is the best alternative.
- Is a completely new ACK frame an alternative ?
- How do STOP_WAITING frames affect the ECN echo overhead.
- Outline possible connection migration actions
- Are there any security implications with the smalle ECN negotiation frame ?

5. Security Considerations

T.B.D

6. Acknowledgements

The following persons have contributed with comments and suggestions for improvements: Mirja Kuehlewind, Koen De Schepper, Piers O'Hanlon, Michael Welzl, Marcelo Bagnulo Braun, Martin Duke

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

7.2. Informative References

[Bagnulo] "Adding Explicit Congestion Notification (ECN) to TCP control packets and TCP retransmissions", <https://tools.ietf.org/id/draft-bagnulo-tcpm-generalized-ecn-00.txt>.

[ECN-fallback] "A Mechanism for ECN Path Probing and Fallback", <https://www.ietf.org/archive/id/draft-kuehlewind-tcpm-ecn-fallback-01.txt>.

[I-D.ietf-aqm-ecn-benefits] Fairhurst, G. and M. Welzl, "The Benefits of using Explicit Congestion Notification (ECN)", draft-ietf-aqm-ecn-benefits-08 (work in progress), November 2015.

[I-D.ietf-quic-transport] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.

[I-D.ietf-tsvwg-ecn-experimentation] Black, D., "Explicit Congestion Notification (ECN) Experimentation", draft-ietf-tsvwg-ecn-experimentation-00 (work in progress), December 2016.

[McQuistin-Perkins] ""Is Explicit Congestion Notification usable with UDP?", Proceedings of the ACM Internet Measurement Conference, Tokyo, Japan, October 2015. DOI:10.1145/2815675.2815716", <https://csperkins.org/publications/2015/10/mcquistin2015ecn-udp.pdf>.

[OHanlon] "ECN support in different OS stacks", <https://mailarchive.ietf.org/arch/msg/rmcat/rRKF3PVmFL2zHCp1bOPKimqSsbM>.

[RFC6679] Westerlund, M., Johansson, I., Perkins, C., O'Hanlon, P., and K. Carlberg, "Explicit Congestion Notification (ECN) for RTP over UDP", RFC 6679, DOI 10.17487/RFC6679, August 2012, <http://www.rfc-editor.org/info/rfc6679>.

[RFC6789] Briscoe, B., Ed., Woundy, R., Ed., and A. Cooper, Ed., "Congestion Exposure (ConEx) Concepts and Use Cases", RFC 6789, DOI 10.17487/RFC6789, December 2012, <http://www.rfc-editor.org/info/rfc6789>.

[RFC7560] Kuehlewind, M., Ed., Scheffenegger, R., and B. Briscoe, "Problem Statement and Requirements for Increased Accuracy in Explicit Congestion Notification (ECN) Feedback", RFC 7560, DOI 10.17487/RFC7560, August 2015, <http://www.rfc-editor.org/info/rfc7560>.

Author's Address

Ingemar Johansson
Ericsson AB
Laboratoriegraend 11
Luleaa 977 53
Sweden

Phone: +46 730783289
Email: ingemar.s.johansson@ericsson.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: December 1, 2017

I. Johansson
Ericsson AB

May 30, 2017

ECN support in QUIC
draft-johansson-quic-ecn-03

Abstract

This memo outlines the ECN (Explicit Congestion Notification) support in QUIC. The draft specifies the ECN negotiation and the ECN echo and in addition, different aspects of fallback in case of ECN failure as well as OS specific issues with ECN and monitoring for ECN capability. The intention is that most of the material ends up updating other new or existing QUIC protocol specifications, thus it may be possible that this draft does not warrant a working group status.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on December 1, 2017.

Copyright Notice

Table of Contents

1. Introduction

ECN support in transport protocols is a fundamental feature that should be included in the QUIC specification as a mandatory element. ECN has the key benefit that it allows for non-destructive congestion

notification by network node, i.e packets are marked instead discarded. This is particularly beneficial for realtime applications with requirements on latency, ECN also has the benefit that it provides with a congestion signal that is unambiguous. The benefits with ECN is described in more detail in [I-D.ietf-aqm-ecn-benefits]. The ECN support should be implemented to support both present and future ECN, the latter is outlined in [I-D.ietf-tsvwg-ecn-experimentation], of particular interest is the ability to discriminate between classic ECN and L4S ECN by means of differentiation between the use of the ECT(0) and ECT(1) code points. This draft does however not delve into the details of the congestion control implementation.

2. Elements of ECN support

This draft covers the following aspects of ECN support:

- ECN negotiation
- ECN echo
- ECN bits in the IP header, semantics
- Fallback in case of ECN fault
- OS socket specifics, access to the ECN bits
- Monitoring

2.1. ECN negotiation

ECN support in QUIC needs to be negotiated. The reasons is that network elements may not support ECN and may either clear the ECN bits or simply discard packets that have the ECN bits set. In addition, a QUIC implementation may not have access to the ECN bits in the IP header due to OS dependent restrictions, investigations (Piers O'Hanlon) have indicated that this is in certain cases an asymmetric property, for instance while it is possible to set the ECN bits it is not possible to read them.

It is also required that the ECN negotiation does not interfere with the connection setup, in other words a failed ECN negotiation should not cause an extra roundtrip for the connection setup.

The suggested method in this draft is to send an ECN negotiation frame when connection setup is completed. Both peers MUST transmit the ECN negotiation frame. The ECN negotiation frame is shown below.

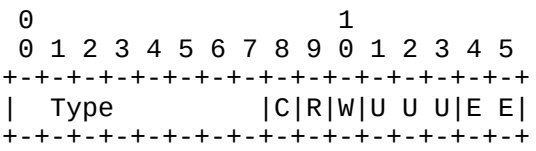


Figure 1: ECN negotation frame

The 2nd byte contains the flags:

- C: Challenge bit, indicates that the transmitted ECN negotiation frame is a challenge, if bit is not set then it is a response.
- R: Possible to read ECN bits in IP header
- W: Possible to write ECN bits in IP header
- EE : Echo of ECN bits
- U: Unused

The ECN negotiation has two steps.

- Challenge/response
- Determine degree of ECN support

2.1.1. Challenge/Response

A peer transmits the ECN negotiation frame with the R,W and EE bits in the 2nd byte set to '0' and the C bit set to '1'. This frame is echoed back with the flags set according to the degree of ECN support and with the ECN bits in the IP header of the received ECN negotiation frame copied to the EE field, the C bit is '0'. As both peers MUST transmit an ECN negotiation frame there will be a total of 4 ECN negotiation frames transmitted, two challenges and two responses.

An ECN negotiation frame should be transmitted in a unique packet, this to avoid that possible loss of ECN negotiation packets cause loss of other frames than the ECN negotiation frame.

The IP header for the ECN negotiation frame should set the ECN bits to CE '11'. When the corresponding response is received then an EE pattern of '11' indicates that ECN is likely supported in the network. This does not give a full guarantee that ECN is supported in the network. Monitoring of the ECN field in the ACK-frame serves to give further indication of ECN support once ECN is turned on.

An ECN negotiation is declared successful when an ECN negotiation response is received that indicates ECN support. A peer is not allowed to set ECT on outgoing data packets until a successful ECN negotiation is done. In other words it is only the ECN negotiation frame that is allowed to set the ECN bits in the IP header until ECN negotiation is concluded and successful.

A lack of an ECN negotiation response may indicate that the ECN challenge frame or the ECN response frame was lost or that a node in the network deliberately discards ECN-CE marked packets. The peer should transmit an additional ECN challenge within an RTO interval in case a negotiation response is not received, a maximum of retransmissions are attempted.

A failed challenge/response phase indicates that ECN should not be used in the connection. [NOTE, a special case is where one peer does not receive an ECN negotiation response but still receives ECT and CE marked packets from the other peer. It is T.B.D how this should be handled]

2.1.2. Determine degree of ECN support

If the ECN challenge/response is successful, the degree of ECN capability depends on how the R, W and EE bits are set.

- R=’1’ and EE= ’11’: It is possible to set the ECN bits in outgoing packets.
- R=’0’ or EE <> ’11’: ECN support is not certain as it is either not possible for remote peer to read the ECN bits or that the ECN bits are altered.
- W=’1’ : It is meaningful to send ECN feedback
- W=’0’ : It is not meaningful to send ECN feedback as the remote peer cannot set (write) the ECN bits in the IP header.

The mode mechanism in [RFC6679] can serve as in input to a solution for the support of ECN in the case that OS ECN support is asymmetric. It is however unclear how a QUIC implementation can determine asymmetric ECN support in the underlying OS. For instance the method to send ECN marked packets to the local host to determine OS support does not reveal if the OS ECN support is asymmetric.

2.2. ECN bits in the IP header, semantics

The ECN bits in the IP header should be set according to the recommendations in [I-D.ietf-tsvwg-ecn-experimentation]. This means that the meaning of ECT(0) and ECT(1) differ.

2.3. ECN echo

The ECN echo should go into the ACK frame [I-D.ietf-quic-transport], this is beneficial as the ECN information can then use some of the already existing data in the ACK frame for improved efficiency.

The proposed alternative use one byte to encode how many bits that encode each of the ECT/CE fields.

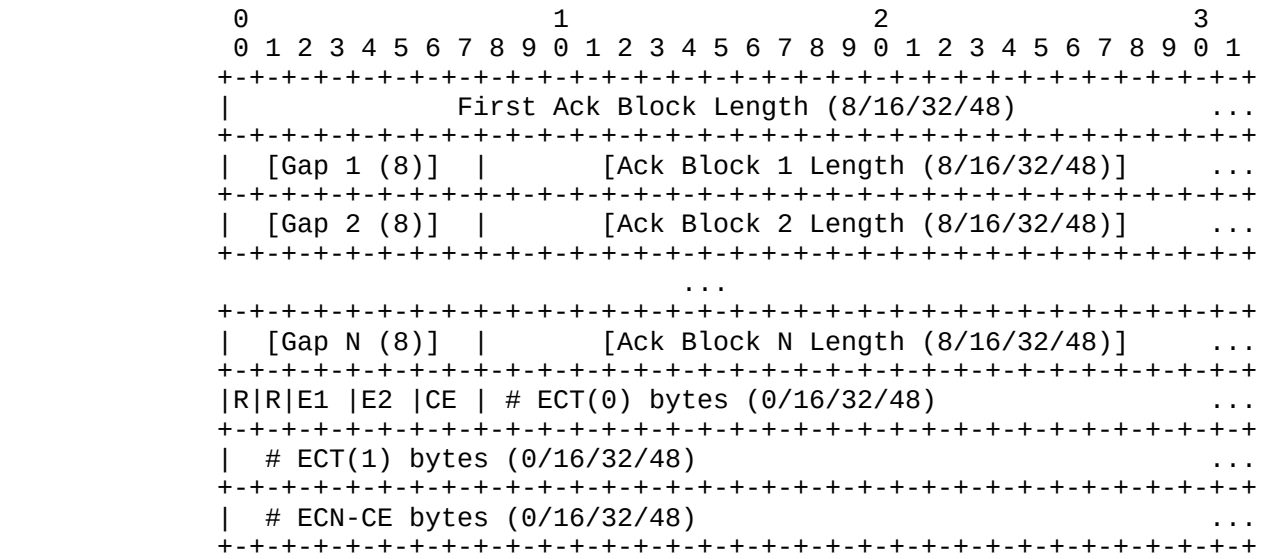


Figure 2: ECN field in ACK frame ACK block

The E1,E2 and CE fields indicate the length of each encoding for the number of ECT(0), ECT(1) and ECN-CE marked bytes. This is encoded as:

- 00: 0 bits
- 01: 16 bits
- 10: 32 bits
- 11: 48bits

R indicates reserved bits.

The proposed encoding enables flexible encoding of the ECN information, with a minimal 1 octet overhead for the cases where ECN is not supported by the connection.

2.4. Fallback in case of ECN fault

ECN can be subject to issues in network equipment, such as remarking to Not-ECN, remarking from ECT(0) to ECT(1) and vice versa or constant remarking to ECN-CE. Furthermore ECT marked packets may be discarded in the network. While these problems seem to be rare, see for instance [McQuistin-Perkins]and [APPLE-ECN], it is still necessary to safeguard against such problems.

A peer should disable ECN for its outgoing packets if ECN fault is detected, it is however still possible for the other peer to use ECN.

TODO add more information as regards to how to detect network ECN faults. [ECN-fallback](expired) gives a few examples for fault detection. Examples on how to detect ECN faults include for instance the method to set ECT and CE for outgoing packets according to a given pattern.

Fallback in case of ECN faults is not an issue only for QUIC, it is here suggested that mechanisms for this is described in a non QUIC related draft, for instance in TSVWG.

2.5. OS socket specifics, access to the ECN bits

ECN support in QUIC comes with the additional challenge that it is necessary to somehow access the ECN bits in the IP headers. In TCP this is provided without major concerns as TCP is generally implemented in OS kernel space. QUIC can however be implemented both in user space or kernel space and is layered on top of UDP, which means that access to the ECN bits is not a given, instead various tricks are needed.

The text below is copy-pasted from [OHanlon].

"To set ECN on Linux, BSD and OSX one can use IP_TOS socket option, with the setsockopt() call, to set the relevant ECN bits of the TOS byte. On Windows one can use a similar technique though firstly one has to enable TOS byte setting by enabling a particular Registry key (DisableUserTOSSetting=0 (see https://msdn.microsoft.com/en-us/library/windows/desktop/dd874008%28v=vs.85%29.aspx One could also probably use the libpcap write functionality."

"To obtain the ECN bits from a packet one needs a mechanism to retrieve the ECN bits from each packet. On Linux, one needs to firstly set the IP_RECVTOS socket option on the receiving socket, and use the recvmsg() call to receive a packet, and then retrieve the TOS byte from the associated csmg structure returned by the recvmsg() call. This still works with linux-4.2.3. On OSX/BSD there are no suitable socket options to retrieve the ECN/TOS bits and one cannot use raw sockets as they do not function for UDP/TCP sockets (they do work with ICMP), so one has to use alternatives such the bpf interface, or a REDIRECT socket. Whilst on Windows it seems that the only way to retrieve the ECN bits is via a raw socket, or custom NDIS driver, though it’s possible there’s an API I’m missing."

TODO: Write a more detailed description on how to implement ECN support in QUIC for different OS stacks.

2.6. Monitoring

A QUIC implementation should monitor the ECN functionality in order to provide input to e.g. service providers to improve ECN support in the networks. Items of interest are:

- Black holes, ECT or CE marked packets are discarded.
- Faulty remarking, e.g. ECT(0) is remarked to ECT(1) or Not-ECT.
- Continuous CE marking, possible indication of faulty on/off ECN marking, but can also be an effect of severe congestion.
- Degree of L4S support. L4S should generally give low queue latency. Estimation of one way queue delay for L4S enabled QUIC connections can be used to determine if there are congested nodes along the path that are not L4S capable.

3. IANA Considerations

T.B.D.

4. Open questions

A list of open questions:

- Is it sufficient that one peer sends an ECN negotiation challenge frame?.
- Should all packets be ECT or should there be special patterns to improve fault detection.

- Write up a more detailed description on how to implement ECN support in QUIC for different OS stacks.
- Is a completely new ACK frame an alternative ?
- Should amount on ECT(0), ECT(1) and CE marked bytes account for the IP+UDP headers or is it only the QUIC header + data that counts ?
- Outline possible connection migration actions
- Are there any security implications with the small ECN negotiation frame ?

5. Security Considerations

T.B.D

6. Acknowledgements

The following persons have contributed with comments and suggestions for improvements: Mirja Kuehlewind, Koen De Schepper, Piers O'Hanlon, Michael Welzl, Marcelo Bagnulo Braun, Martin Duke

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

7.2. Informative References

[APPLE-ECN] Apple Inc., "TCP ECN: Experience with Enabling ECN on the Internet", <https://www.ietf.org/proceedings/98/slides/slides-98-maprg-tcp-ecn-experience-with-enabling-ecn-on-the-internet-padma-bhooma-00.pdf>.

[Bagnulo] "Adding Explicit Congestion Notification (ECN) to TCP control packets and TCP retransmissions", <https://tools.ietf.org/id/draft-bagnulo-tcpm-generalized-ecn-00.txt>.

[ECN-fallback]
"A Mechanism for ECN Path Probing and Fallback", <https://www.ietf.org/archive/id/draft-kuehlewind-tcpm-ecn-fallback-01.txt>.

[I-D.ietf-aqm-ecn-benefits]
Fairhurst, G. and M. Welzl, "The Benefits of using Explicit Congestion Notification (ECN)", draft-ietf-aqm-ecn-benefits-08 (work in progress), November 2015.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-03 (work in progress), May 2017.

[I-D.ietf-tsvwg-ecn-experimentation]
Black, D., "Explicit Congestion Notification (ECN) Experimentation", draft-ietf-tsvwg-ecn-experimentation-02 (work in progress), April 2017.

[McQuistin-Perkins]
""Is Explicit Congestion Notification usable with UDP?", Proceedings of the ACM Internet Measurement Conference, Tokyo, Japan, October 2015. DOI:10.1145/2815675.2815716", <https://csperkins.org/publications/2015/10/mcquistin2015ecn-udp.pdf>.

[OHanlon] "ECN support in different OS stacks", <https://mailarchive.ietf.org/arch/msg/rmcat/rRKF3PVmFL2zHCp1bOPKimqSsbM>.

[RFC6679] Westerlund, M., Johansson, I., Perkins, C., O'Hanlon, P., and K. Carlberg, "Explicit Congestion Notification (ECN) for RTP over UDP", RFC 6679, DOI 10.17487/RFC6679, August 2012, <http://www.rfc-editor.org/info/rfc6679>.

[RFC6789] Briscoe, B., Ed., Woundy, R., Ed., and A. Cooper, Ed., "Congestion Exposure (ConEx) Concepts and Use Cases", RFC 6789, DOI 10.17487/RFC6789, December 2012, <http://www.rfc-editor.org/info/rfc6789>.

[RFC7560] Kuehlewind, M., Ed., Scheffenegger, R., and B. Briscoe, "Problem Statement and Requirements for Increased Accuracy in Explicit Congestion Notification (ECN) Feedback", RFC 7560, DOI 10.17487/RFC7560, August 2015, <http://www.rfc-editor.org/info/rfc7560>.

Author's Address

Ingemar Johansson
Ericsson AB
Laboratoriegraend 11
Luleaa 977 53
Sweden

Phone: +46 730783289
Email: ingemar.s.johansson@ericsson.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 9, 2017

M. Kuehlewind
B. Trammell
ETH Zurich
March 08, 2017

Applicability of the QUIC Transport Protocol
draft-kuehlewind-quic-applicability-00

Abstract

This document discusses the applicability of the QUIC transport protocol, focusing on caveats impacting application protocol development and deployment over QUIC. Its intended audience is designers of application protocol mappings to QUIC, and implementors of these application protocols.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 9, 2017.

Table of Contents

1. Introduction

QUIC [I-D.ietf-quic-transport] is a new transport protocol currently under development in the IETF quic working group, focusing on support of semantics as needed for HTTP/2 [I-D.ietf-quic-http] such as stream-multiplexing to avoid head-of-line blocking. Based on current deployment practices, QUIC is encapsulated in UDP and encrypted by default. This means the version of QUIC that is currently under development will integrate TLS 1.3 [I-D.ietf-quic-tls] to encrypt all payload data and most header information.

This document provides guidance for application developers that want to use the QUIC protocol without implementing it on their own. This includes general guidance for application use of HTTP/2 over QUIC as well as the use of other application layer protocols over QUIC. For specific guidance on how to integrate HTTP/2 with QUIC, see [I-D.ietf-quic-http].

In the following sections we discuss specific caveats to QUIC's applicability, and issues that application developers must consider when using QUIC as a transport for their application.

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when these words are capitalized, they have a special meaning as defined in [RFC2119].

2. The Necessity of Fallback

QUIC uses UDP as a substrate for userspace implementation and port numbers for NAT and middlebox traversal. While there is no evidence of widespread, systematic disadvantage of UDP traffic compared to TCP in the Internet [Edeline16], somewhere between three [Trammell16] and five [Swett16] percent of networks simply block UDP traffic. All applications running on top of QUIC must therefore either be prepared to accept connectivity failure on such networks, or be engineered to fall back to some other transport protocol. This fallback SHOULD provide TLS 1.3 or equivalent cryptographic protection, if available, in order to keep fallback from being exploited as a downgrade attack. In the case of HTTP, this fallback is TLS 1.3 over TCP.

These applications must operate, perhaps with impaired functionality, in the absence of features provided by QUIC not present in the fallback protocol. For fallback to TLS over TCP, the most obvious difference is that TCP does not provide stream multiplexing and therefore stream multiplexing would need to be implemented in the application layer if needed. Further, TCP by default does not support 0-RTT session resumption. TCP Fast Open could be used, but might no be supported by the far end or could be blocked on the network path. Note that there is some evidence of middleboxes blocking SYN data even if TFO was successfully negotiated (see [PaaschNanog]). Moreover, while encryption (in this case TLS) is inseparable integrated with QUIC, TLS negotiation over TCP can be blocked. In case it is RECOMMENDED to abort the connection, allowing the application to present a suitable prompt to the user that secure communication is unavailable.

We hope that the deployment of a proposed standard version of the QUIC protocol will provide an incentive for these networks to permit QUIC traffic. Indeed, the ability to treat QUIC traffic statefully as discussed in section 3.1 of [draft-kuehlewind-quic-manageability] would remove one network management incentive to block this traffic.

3. Zero RTT: Here There Be Dragons

QUIC provides for 0-RTT connection establishment (see section 3.2 of [I-D.ietf-quic-transport]). However, data in the frames contained in the first packet of a such a connection must be treated specially by the application layer. Since a retransmission of these frames resulting from a lost acknowledgment may cause the data to appear twice, either the application-layer protocol has to be designed such that all such data is treated as idempotent, or there must be some application-layer mechanism for recognizing spuriously retransmitted frames and dropping them.

Applications that cannot treat data that may appear in a 0-RTT connection establishment as idempotent MUST NOT use 0-RTT establishment. For this reason the QUIC transport SHOULD provide an interface for the application to indicate if 0-RTT support is in general desired or a way to indicate if data is idempotent.

4. Stream versus Flow Multiplexing

QUIC's stream multiplexing feature allows applications to run multiple streams over a single connection, without head-of-line blocking between streams, associated at a point in time with a single five-tuple. Streams are meaningful only to the application; since stream information is carried inside QUIC's encryption boundary, no information about the stream(s) whose frames are carried by a given packet is visible to the network.

Stream multiplexing is not intended to be used for differentiating streams in terms of network treatment. Application traffic requiring different network treatment SHOULD therefore be carried over different five-tuples (i.e. multiple QUIC connections). Given QUIC's ability to send application data on the first packet of a connection (if a previous connection to the same host has been successfully established to provide the respective credentials), the cost for establishing another connection are extremely low.

[EDITOR'S NOTE: For discussion: If establishing a new connection does not seem to be sufficient, the protocol's rebinding functionality (see section 3.7 of [I-D.ietf-quic-transport]) could be extended to allow multiple five-tuples to share a connection ID simultaneously, instead of sequentially.]

5. Prioritization

Stream prioritization is not exposed to the network, nor to the receiver. Prioritization can be realized by the sender and the QUIC transport should provide and interface for applications to prioritize streams [I-D.ietf-quic-transport].

Priority handling of retransmissions may be implemented in the transport layer and [I-D.ietf-quic-transport] does not specify a specific way how this must be handled. Currently QUIC only provides fully reliable stream transmission, and as such prioritization of retransmission is likely beneficial. For not fully reliable streams priority scheduling of retransmissions over data of higher-priority streams might not be desired. In this case QUIC could also provide an interface or derive the prioritization decision from the reliability level of the stream.

6. Graceful connection closure

[EDITOR'S NOTE: give some guidance here about the steps an application should take; however this is still work in progress]

7. Information exposure and the Connection ID

QUIC exposed some information to the network in the unencrypted part of the header. This is either because there is no encryption context established yet or because this information is intended to be consumed by the network. Some of these information can be optionally exposed (still under discussion). Given that exposing these information can have privacy implications, an application may indicate to not support exposure of certain information.

In case of the connection ID this can be the case if the application has additional information that the client is not behind a NAT and the server is not behind a load balancer, and therefore it is unlikely that the addresses will be re-binded.

8. Use of Versions and Cryptographic Handshake

Versioning in QUIC may change the whole protocol behavior, beside some header fields that have been declared to be fixed. As such a new or higher version of QUIC does not necessarily provide a better service but just a very different service, an application needs to be able to select which versions of QUIC it wants to use.

The use of a different encryption scheme than TLS1.3 or higher needs a new version of QUIC. [I-D.ietf-quic-transport] specifies requirements for the cryptographic handshake as currently realized by TLS1.3 and described in a separate specification [I-D.ietf-quic-tls]. This split is performed to enable light-weight versioning with different cryptographic handshakes.

9. IANA Considerations

This document has no actions for IANA.

10. Security Considerations

See the security considerations in [I-D.ietf-quic-transport] and [I-D.ietf-quic-tls]; the security considerations for the underlying transport protocol are relevant for applications using QUIC, as well.

Application developers should note that any fallback they use when QUIC cannot be used due to network blocking of UDP SHOULD guarantee the same security properties as QUIC; if this is not possible, the

connection SHOULD fail to allow the application to explicitly handle fallback to a less-secure alternative. See Section 2.

11. Acknowledgments

This work is partially supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. This support does not imply endorsement.

12. References

12.1. Normative References

[I-D.ietf-quic-tls]
Thomson, M. and S. Turner, "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-01 (work in progress), January 2017.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

12.2. Informative References

[draft-kuehlewind-quic-manageability]
Kuehlewind, M. and B. Trammell, "Manageability of the QUIC Transport Protocol", March 2017.

[Edeline16]
Edeline, K., Kuehlewind, M., Trammell, B., Aben, E., and B. Donnet, "Using UDP for Internet Transport Evolution (arXiv preprint 1612.07816)", December 2016.

[I-D.ietf-quic-http]
Bishop, M., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-01 (work in progress), January 2017.

[PaaschNanog]
Paasch, C., "Network Ssupport for TCP Fast Open (NANOG 67 presentation)", June 2016.

[Swett16] Swett, I., "QUIC Deployment Experience at Google (IETF96 QUIC BoF presentation)", July 2016.

[Trammell16]
Trammell, B. and M. Kuehlewind, "Internet Path Transparency Measurements using RIPE Atlas (RIPE72 MAT presentation)", May 2016.

Authors' Addresses

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: mirja.kuehlewind@tik.ee.ethz.ch

Brian Trammell
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 10, 2017

M. Kuehlewind
B. Trammell
ETH Zurich
D. Druta
AT&T

March 09, 2017

Manageability of the QUIC Transport Protocol
draft-kuehlewind-quic-manageability-00

Abstract

This document discusses manageability of the QUIC transport protocol, focusing on caveats impacting network operations involving QUIC traffic. Its intended audience is network operators, as well as content providers that rely on the use of QUIC-aware middleboxes, e.g. for load balancing.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 10, 2017.

Table of Contents

1. Introduction

QUIC [I-D.ietf-quic-transport] is a new transport protocol currently under development in the IETF quic working group, focusing on support of semantics as needed for HTTP/2 [I-D.ietf-quic-http]. Based on current deployment practices, QUIC is encapsulated in UDP and encrypted by default. The current version of QUIC integrates TLS [I-D.ietf-quic-tls] to encrypt all payload data and most header information. Given QUIC is an end-to-end transport protocol, all information in the protocol header, even that which can be inspected, is is not meant to be mutable by the network, and will therefore be integrity-protected to the extent possible.

This document provides guidance for network operation on the management of QUIC traffic. This includes guidance on how to interpret and utilize information that is exposed by QUIC to the network as well as explaining requirement and assumptions that the QUIC protocol design takes toward the expected network treatment. It also discusses how common network management practices will be impacted by QUIC.

Of course, network management is not a one-size-fits-all endeavour: practices considered necessary or even mandatory within enterprise networks with certain compliance requirements, for example, would be impermissible on other networks without those requirements. This document therefore does not make any specific recommendations as to which practices should or should not be applied; for each practice, it describes what is and is not possible with the QUIC transport protocol as defined.

QUIC is at the moment very much a moving target. This document refers the state of the QUIC working group drafs as well as to changes under discussion, via issues and pull requests in GitHub current as of the time of writing.

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It’s not shouting; when these words are capitalized, they have a special meaning as defined in [RFC2119].

2. Features of the QUIC Wire Image

In this section, we discusses those aspects of the QUIC transport protocol that have an impact on the design and operation of devices that forward QUIC packets. Here, we are concerned primarily with QUIC’s unencrypted wire image, which we define as the information available in the packet header in each QUIC packet, and the dynamics of that information. Since QUIC is a versioned protocol, everything about the header format can change except the mechanism by which a receiver can determine whether and where a version number is present, and the meaning of the fields used in the version negotiation process. This document is focused on the protocol as presently defined in [I-D.ietf-quic-transport] and [I-D.ietf-quic-tls], and will change to track those documents.

2.1. QUIC Packet Header Structure

The QUIC packet header is under active development; see section 5 of [I-D.ietf-quic-transport] for the present header structure, and https://github.com/quicwg/base-drafts/pull/361 for one current proposed redesign.

Currently the first bit of the QUIC header indicates the present of a long header that exposed more information than the short. The long header is typically used during connection start or for other control processes while the short header will be used on mostly packets to limited unnecessary header overhead. The following information may be exposed in the packet header:

- version number: The version number is present during version negotiation.

- connection ID: The connection ID identifies the connection associated with a QUIC packet, for load-balancing and NAT rebinding purposes; see Section 2.3.

- packet number: Every packet has an associated packet number; this packet number increases with each packet, and the least-significant bits of the packet number are present on each packet; see Section 2.4.

- public reset indication: Public reset packets expose the fact that a connection is being torn down to devices along the path. The applicability of public reset is currently under discussion; see https://github.com/quicwg/base-drafts/issues/353 and https://github.com/quicwg/base-drafts/pull/20.

- key phase: To support 0-RTT session establishment, QUIC uses two key phases; the key phase of each packet must be exposed to support efficient reception.

- additional flags: Additional flags for diagnostic use are also under consideration; see https://github.com/quicwg/base-drafts/issues/279.

[Editor’s note: also further discuss which bits cannot change with versioning]

2.2. Integrity Protection of the Wire Image

As soon as the cryptograhic context is established, all information in the QUIC header, including that exposed in the packet header, is integrity protected. Therefore, devices on path MUST NOT change QUIC packet headers, as alteration of header information would cause packet drop due to a failed integrity check at the receiver.

2.3. Connection ID and Rebinding

The connection ID in the QUIC packer header is used to allow routing of QUIC packets at load balancers on other than five-tuple information, ensuring that related flows are appropriately balanced together; and to allow rebinding of a connection after one of the endpoint’s addresses changes - usually the client’s, in the case of the HTTP binding. The connection ID is proposed by the server during connection establishment. A flow might change one of its IP addresses but keep the same connection ID, as noted in Section 2.1, and the connection ID may change during a connection as well; see

section 6.3 of [I-D.ietf-quic-transport]. See also https://github.com/quicwg/base-drafts/issues/349 for ongoing discussion of the Connection ID.

2.4. Packet Numbers

The packet number field is always present in the QUIC packet header. The packet number exposes the least significant 32, 16, or 8 bits of an internal packet counter per flow direction that increments with each packet sent. This packet counter is initialized with a random 31-bit initial value at the start of a connection.

Unlike TCP sequence numbers, this packet number increases with every packet, including those containing only acknowledgment or other control information. Indeed, whether a packet contains user data or only control information is intentionally left unexposed to the network.

While loss detection in QUIC is based on packet numbers, congestion control by default provides richer information than vanilla TCP does. Especially, QUIC does not rely on duplicated ACKs, making it more tolerant of packet re-ordering.

2.5. Greasing

[Editor’s note: say something about greasing if added to the transport draft]

3. Specific Network Management Tasks

In this section, we address specific network management and measurement techniques and how QUIC’s design impacts them.

3.1. Stateful Treatment of QUIC Traffic

Stateful network devices such as firewalls use exposed header information to support state setup and tear-down. [I-D.trammell-plus-statefulness] provides a general model for in-network state management on these devices, independent of transport protocol. Features already present in QUIC may be used for state maintenance in this model. Here, there are two important goals: distinguishing valid QUIC connection establishment from other traffic, in order to establish state; and determining the end of a QUIC connection, in order to tear that state down.

1-RTT connection establishment, using a TLS handshake on stream 0, is detectable using heuristics similar to those used to detect TLS over TCP. 0-RTT connection establishment, however, provides no particular

heuristic for differentiation from random background traffic at this time.

Exposure of connection shutdown is currently under discussion; see https://github.com/quicwg/base-drafts/issues/353 and https://github.com/quicwg/base-drafts/pull/20.

3.2. Measurement of QUIC Traffic

Passive measurement of TCP performance parameters is commonly deployed in access and enterprise networks to aid troubleshooting and performance monitoring without requiring the generation of active measurement traffic.

The presence of packet numbers on most QUIC packets allows the trivial one-sided estimation of packet loss and reordering between the sender and a given observation point. However, since retransmissions are not identifiable as such, loss between an observation point and the receiver cannot be reliably estimated.

The lack of any acknowledgement information or timestamping information in the QUIC packet header makes running passive estimation of latency via round trip time (RTT) impossible. RTT can only be measured at connection establishment time, and only when 1-RTT establishment is used.

Note that adding packet number echo (as in https://github.com/quicwg/base-drafts/pull/367 or https://github.com/quicwg/base-drafts/pull/368) to the public header would allow passive RTT measurement at on-path observation points. For efficiency purposes, this packet number echo need not be carried on every packet, and could be made optional, allowing endpoints to make a measurability/efficiency tradeoff; see section 4 of [IPIM]. Note further that this facility would have significantly better measurability characteristics than sequence-acknowledgement-based RTT measurement currently available in TCP on typical asymmetric flows, as adequate samples will be available in both directions, and packet number echo would be decoupled from the underlying acknowledgment machinery; see e.g. [Ding2015]

Note in-network devices can inspect and correlate connection IDs for partial tracking of mobility events.

3.3. DDoS Detection and Mitigation

For enterprises and network operators one of the biggest management challenges is dealing with Distributed Denial of Service (DDoS) attacks. Some network operators offer Security as a Service (SaaS)

solutions that detect attacks by monitoring, analyzing and filtering traffic. These approaches generally utilize network flow data [RFC7011]. If any flows pose a threat, usually they are routed to a "scrubbing environment" where the traffic is filtered, allowing the remaining "good" traffic to continue to the customer environment.

This type of DDoS mitigation is fundamentally based on tracking state for flows (see Section 3.1) that have receiver confirmation and a proof of return-routability, and classifying flows as legitimate or DoS traffic. The QUIC packet header currently does not support an explicit mechanism to easily distinguish legitimate QUIC traffic from other UDP traffic. However, the first packet in a QUIC connection will usually be a client cleartext packet with a version field and a connection ID. This can be used to identify the first packet of the connection (also see https://github.com/quicwg/base-drafts/issues/185).

If the QUIC handshake was not observed by the defense system, the connection ID can be used as a confirmation signal as per [I-D.trammell-plus-statefulness]. In this case, similar as for all in-network functions that rely on the connection ID, a defense system can only rely on this signal for known QUIC's versions and if the connection ID is present (also see https://github.com/quicwg/base-drafts/issues/293).

Further, the use of a connection ID to support connection migration renders 5-tuple based filtering insufficient, and requires more state to be maintained by DDoS defense systems. However, it is questionable if connection migrations needs to be supported in a DDOS attack or if a defense system might simply rely on the fast resumption mechanism provided by QUIC. This problem is also related to these issues under discussion: https://github.com/quicwg/base-drafts/issues/203 and https://github.com/quicwg/base-drafts/issues/349

3.4. QoS support and ECMP

QUIC does not provide any additional information on requirements on Quality of Service (QoS) provided from the network. QUIC assumes that all packets with the same 5-tuple {dest addr, source addr, protocol, dest port, source port} will receive similar network treatment. That means all stream that are multiplexed over the same QUIC connection require the same network treatment and are handled by the same congestion controller. If differential network treatment is desired, multiple QUIC connection to the same server might be used, given that establishing a new connection using 0-RTT support is cheap and fast.

QoS mechanisms in the network MAY also use the connection ID for service differentiation as usually a change of connection ID is bind to a change of address which anyway is likely to lead to a re-route on a different path with different network characteristics.

Given that QUIC is more tolerant of packet re-ordering than TCP (see Section 2.4), Equal-cost multi-path routing (ECMP) does not necessarily need to be flow based. However, 5-tuple (plus eventually connection ID if present) matching is still beneficial for QoS given all packets are handled by the same congestion controller.

3.5. Load balancing

[Editor's note: explain how this works as soon as we have decided who chooses the connection ID and when to set it. Related to https://github.com/quicwg/base-drafts/issues/349]

4. IANA Considerations

This document has no actions for IANA.

5. Security Considerations

Supporting manageability of QUIC traffic inherently involves tradeoffs with the confidentiality of QUIC's control information; this entire document is therefore security-relevant.

Some of the properties of the QUIC header used in network management are irrelevant to application-layer protocol operation and/or user privacy. For example, packet number exposure (and echo, as proposed in this document), as well as connection establishment exposure for 1-RTT establishment, make no additional information about user traffic available to devices on path.

At the other extreme, supporting current traffic classification methods that operate through the deep packet inspection (DPI) of application-layer headers are directly antithetical to QUIC's goal to provide confidentiality to its application-layer protocol(s); in these cases, alternatives must be found.

6. Acknowledgments

This work is partially supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. This support does not imply endorsement.

7. References

7.1. Normative References

[I-D.ietf-quic-tls]
Thomson, M. and S. Turner, "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-01 (work in progress), January 2017.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

7.2. Informative References

[Ding2015]
Ding, H. and M. Rabinovich, "TCP Stretch Acknowledgments and Timestamps - Findings and Impliciations for Passive RTT Measurement (ACM Computer Communication Review)", July 2015.

[draft-kuehlewind-quic-applicability]
Kuehlewind, M. and B. Trammell, "Applicability of the QUIC Transport Protocol", March 2017.

[I-D.ietf-quic-http]
Bishop, M., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-01 (work in progress), January 2017.

[I-D.trammell-plus-statefulness]
Kuehlewind, M., Trammell, B., and J. Hildebrand, "Transport-Independent Path Layer State Management", draft-trammell-plus-statefulness-02 (work in progress), December 2016.

[IPIM] Allman, M., Beverly, R., and B. Trammell, "In-Protocol Internet Measurement (arXiv preprint 1612.02902)", December 2016.

[RFC7011] Claise, B., Ed., Trammell, B., Ed., and P. Aitken, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information", STD 77, RFC 7011, DOI 10.17487/RFC7011, September 2013, <http://www.rfc-editor.org/info/rfc7011>.

Authors' Addresses

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: mirja.kuehlewind@tik.ee.ethz.ch

Brian Trammell
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

Dan Druta
AT&T

Email: dd5826@att.com